Simplification of Cylindrical Algebraic Formulas

Changbo Chen[†], Marc Moreno Maza[†,‡]

[†] Chongqing Key Laboratory of Automated Reasoning and Cognition, Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences
[‡] ORCCA, University of Western Ontario
changbo.chen@hotmail.com, moreno@csd.uwo.ca

Abstract. For a set S of cells in a cylindrical algebraic decomposition of $\mathbb{R}^n$, we introduce the notion of generalized cylindrical algebraic formula (GCAF) associated with S. We propose a multi-level heuristic algorithm for simplifying the cylindrical algebraic formula associated with S into a GCAF. The heuristic strategies are motivated by solving examples coming from the application of automatic loop transformation. While the algorithm works well on these examples, its effectiveness is also illustrated by examples from other application domains.

1 Introduction

Cylindrical algebraic decomposition (CAD), introduced by G. E. Collins [8], is a fundamental tool in real algebraic geometry. One of its main applications, also the initial motivation, is to solve quantifier elimination problems in the first order theory of real closed fields. Since its introduction, CAD has been improved by many authors and applied to numerous applications. The implementation of CAD is now available in different software, such as QEPCAD, Mathematica, Redlog, SyNRAC, `RegularChains`, and many others.

A CAD of $\mathbb{R}^n$ decomposes $\mathbb{R}^n$ into disjoint connected semi-algebraic sets, called cells, such that any two cells are cylindrically arranged, which implies that the projections of any two cells onto any $\mathbb{R}^k$, $1 \leq k < n$, are either identical or disjoint. For a given semi-algebraic set S, one can compute a CAD $\mathcal{C}$ such that S can be written as a union of cells in $\mathcal{C}$. Each cell is represented by a cylindrical algebraic formula (CAF) [14], whose zero set is the cell.

The CAF $\phi_c(x_1, \ldots, x_n)$ representing a CAD cell c of $\mathbb{R}^n$ is a conjunction of finitely many atomic formulas of the form $x_i \ \sigma \ \mathrm{Root}_{x_i,k}(p)$, where $p \in \mathbb{R}[x_1, \ldots, x_i]$ and $\mathrm{Root}_{x_i,k}(p)$ denotes the k-th real root (counting multiplicities) of p treated as a univariate polynomial in x_i. The precise definition of CAF is given in Section 2. The CAF $\phi_c(x_1, \ldots, x_n)$ has a very nice property, namely the projection of c onto $\mathbb{R}^j$, $1 \leq j < n$, is exactly the zero set of the sub-formula of ϕ_c, which is obtained by taking the conjunction of all atomic formulas in ϕ_c involving only the variables $x_1, \ldots, x_j$. Let S be a set of cells $c_1, \ldots, c_t$ from a CAD $\mathcal{C}$. Denote by ϕ_S the zero set of S, thus we have $\phi_S := \vee_{i=1}^{t} \phi_{c_i}$. The formula ϕ_S is also called a *cylindrical algebraic formula.*

A CAF is a special extended Tarski formula, see [2]. While a Tarski formula is often the default output of quantifier elimination procedures, a CAF is also important for several reasons. Firstly, computing CAFs can be done by means of a CAD procedure without introducing additional augmented projection factors, which can bring substantial savings in terms of computation resources. Secondly, CAFs have a nice structure: the projection of a CAF onto any lower-dimensional space can be easily read off from the CAF itself, as mentioned before. Moreover, since a CAF is used to describe CAD cells, it naturally exhibits a polychotomous structure. This property is usually not true for Tarski formula output. Thirdly, each atomic formula of a CAF has the convenient format $x\ \sigma\ E$, where E is an indexed root expression. This explicit expression is particularly useful in applications which care about the specific value of each coordinate, like in loop transformations of computer program [12, 11]. Last but not least, performing set-theoretical operations on CAFs can be done efficiently, without explicit conversion to Tarski formulas [14]. This latter property supports an incremental algorithm for computing CADs [15].

While CAFs have many advantages, they have also their own drawbacks. Firstly, indexed root expressions are usually less handy to manipulate than polynomial expressions. This is because a polynomial function is defined everywhere while an indexed root expression is usually defined on a particular set. Secondly, due to numerous CAD cells being generated, a CAD-based QE solver usually outputs very lengthy CAFs, which could make the output formula not easy to use. For the particular application of loop transformation of computer programs, too many case distinctions might substantially increase the arithmetic cost of evaluating the transformed program as well as the number of misses in accessing cache memories. Therefore, simplification of CAFs is clearly needed. However, we have not seen much literature devoted to this topic except Chapter 8 of Brown's PhD thesis [2], The differences between Brown's approach and the one proposed is the present paper are discussed in Section 6. We remark that the `Reduce` command of `Mathematica` does perform some simplification before outputting CAFs. See Section 5 for an experimental comparison with `Mathematica`.

Since CAFs are generated from CAD cells, it is a natural idea to make use of the CAD data structure to simplify CAFs. In this paper, we produce a multi-level merging procedure for simplifying CAFs by exploiting structural properties of the CAD from which those CAFs are being generated. Although this procedure aims at improving the output of CAD solvers based on regular chains, it is also applicable to other CAD solvers. The merging procedure, presented formally in Section 4, consists of several reasonable and workable heuristics, most of which are motivated by solving examples taken from [12, 7]. See Section 3 for details. The simplification procedure has four levels, where an upper level never produces more conjunctive clauses than the lower levels. The first two levels merge adjacent CAD cells, whereas the last two levels attempt to simplify a CAF into a single conjunctive clause, which is usually expected in the application of loop transformation. Thus the first two levels are expected to be effective for general QE problems whereas the last two are expected to be effective for QE

problems arising from loop transformation. This expectation is justified also by the experimentation in Section 5.

The method has been implemented and new options are added to both the CylindricalAlgebraicDecompose and QuantifierElimination commands of the `RegularChains` library. The effectiveness of this algorithm is illustrated by examples in Sections 3 and 5. The experimentation shows that our heuristics work well. The running time overhead of simplification compared to the running time of the quantifier elimination procedure itself is negligible in the first two levels and acceptable in the advanced levels of the proposed heuristics.

There have already been a few works on the simplification of Tarski formulas, see for example [10, 4, 3, 13]. Our work is concerned with the simplification of extended Tarski formulas, which allow indexed root expressions besides polynomial constraints. Moreover, the simplification goal here is to reduce as much as possible the number of conjunctive CAF clauses while still maintaining the feature of case distinctions. We emphasize that the motivation and the main targeting application of this work is to unify the CAFs generated in the application of loop transformation. In such applications, explicit bounds of loop indices are needed and the number of case distinctions is expected to be as small as possible in order to reduce the code size.

2 Preliminary

In this section, we first review the notion of cylindrical algebraic decomposition and cylindrical algebraic formula (CAF). Then we define the notion of generalized CAF in order to represent the combination of CAFs.

Real algebraic function. Let $S \subset \mathbb{R}^n$. Let $f(x_1, \ldots, x_n, y) \in \mathbb{R}[x_1, \ldots, x_n, y]$. Let k be a positive integer. Assume that for every point α of S, the univariate polynomial $f(\alpha, y)$ has at least k real roots ordered by increasing value, counting multiplicities. Let $\mathrm{Root}_{y,k}(f)$ be a function which maps every point α of S to the k-th real root of $f(\alpha, y)$. The function $\mathrm{Root}_{y,k}(f)$ is called a real algebraic function defined on S.

Stack over a semia-algebraic set. Let S be a connected semi-algebraic subset of $\mathbb{R}^{n-1}$. The *cylinder* over S in $\mathbb{R}^n$ is defined as $Z_{\mathbb{R}}(S) := S \times \mathbb{R}$. Let $\theta_1 < \cdots < \theta_s$ be continuous real algebraic functions defined on S. Denote $\theta_0 = -\infty$ and $\theta_{s+1} := \infty$. The intersection of the graph of θ_i with $Z_{\mathbb{R}}(S)$ is called the *θ_i-section* of $Z_{\mathbb{R}}(S)$. The set of points between θ_i-section and θ_{i+1}-section, $0 \leq i \leq s$, of $Z_{\mathbb{R}}(S)$ is a connected semi-algebraic subset of $\mathbb{R}^n$, called a *(θ_i, θ_{i+1})-sector* of $Z_{\mathbb{R}}(S)$. The sequence (θ_0, θ_1)-sector, θ_1-section, (θ_1, θ_2)-sector, …, θ_s-section, (θ_s, θ_{s+1})-sector form a disjoint decomposition of $Z_{\mathbb{R}}(S)$, called a *stack* over S, which is uniquely defined for given functions $\theta_1 < \cdots < \theta_s$.

Cylindrical algebraic decomposition. Let π_{n-1} be the standard projection from $\mathbb{R}^n$ to $\mathbb{R}^{n-1}$ mapping $(x_1, \ldots, x_{n-1}, x_n)$ onto $(x_1, \ldots, x_{n-1})$. A finite partition $\mathcal{D}$ of $\mathbb{R}^n$ is called a *cylindrical algebraic decomposition* (CAD) of $\mathbb{R}^n$ if one of the following properties holds.

– either $n = 1$ and $\mathcal{D}$ is a stack over $\mathbb{R}^0$,
– or the set of $\{\pi_{n-1}(D) \mid D \in \mathcal{D}\}$ is a CAD of $\mathbb{R}^{n-1}$ and each $D \in \mathcal{D}$ is a section or sector of the stack over $\pi_{n-1}(D)$.

When this holds, the elements of $\mathcal{D}$ are called *cells*. The set $\{\pi_{n-1}(D) \mid D \in \mathcal{D}\}$ is called the *induced* CAD of $\mathcal{D}$. A CAD $\mathcal{D}$ of $\mathbb{R}^n$ can be encoded by a tree, called a CAD tree (denoted by T), as below. The root node, denoted by r, is $\mathbb{R}^0$. The children nodes of r are exactly the elements of the stack over $\mathbb{R}^0$. Let T_{n-1} be a CAD tree of the induced CAD of $\mathcal{D}$ in $\mathbb{R}^{n-1}$. For any leaf node C of T_{n-1}, its children nodes are exactly the elements of the stack over C.

Cylindrical algebraic formula. Let c be a cell in a CAD of $\mathbb{R}^n$. A cylindrical algebraic formula associated with c, denoted by ϕ_c, is defined recursively.

(i) The case for $n = 1$. If $c = \mathbb{R}$, then $\phi_c := true$. If c is a point α, then define $\phi_c := x_1 = \alpha$. If c is an open interval $(\alpha, \beta) \neq \mathbb{R}$, then $\phi_c := c > \alpha \wedge c < \beta$. For the special case that $\alpha = -\infty$, then ϕ_c is simply written as $c < \beta$. Similarly if $\beta = +\infty$, ϕ_c is simply written as $c > \alpha$.

(ii) The case for $n > 1$. Let c_{n-1} be the projection of c onto $\mathbb{R}^{n-1}$. If $c = c_{n-1} \times \mathbb{R}$, then define $\phi_c := \phi_{c_{n-1}}$. If c is an θ_i-section, then $\phi_c := \phi_{c_{n-1}} \wedge x_n = \theta_i$. If c is an (θ_i, θ_{i+1})-sector, then $\phi_c := \phi_{c_{n-1}} \wedge x_n > \theta_i \wedge x_n < \theta_{i+1}$. If $\theta_i = -\infty$, then ϕ_c is simply written as $\phi_{c_{n-1}} \wedge x_n < \theta_{i+1}$. If $\theta_{i+1} = +\infty$, then ϕ_c is simply written as $\phi_{c_{n-1}} \wedge x_n > \theta_i$.

If ϕ_c is the CAF associated with c, its zero set is defined as $Z_{\mathbb{R}}(\phi_c) := c$. Let S be a set of disjoint cells in a CAD. If $S = \emptyset$, $\phi_S := false$. Otherwise, a CAF associated with S is defined as $\phi_S := \vee_{c \in S} \phi_c$. Its zero set is $Z_{\mathbb{R}}(\phi_S) := \cup_{c \in S} c$.

Example 1 *Consider the closed unit disk S defined by $x^2 + y^2 \leq 1$. Then a CAF associated with S is as below.*

$$\begin{aligned}(x = -1 \wedge y = 0) \vee\ & (-1 < x \wedge x < 1 \wedge y = -\sqrt{1 - x^2}) \\ \vee\ & (-1 < x \wedge x < 1 \wedge -\sqrt{1 - x^2} < y \wedge y < \sqrt{1 - x^2}) \\ \vee\ & (-1 < x \wedge x < 1 \wedge y = \sqrt{1 - x^2}) \\ \vee\ & (x = 1 \wedge y = 0)\end{aligned}$$

Extended Tarski formula [2]. A (restricted) *extended Tarski formula* (ETF) is a Tarski formula, with possibly the addition of atomic formulas of the form $x_i\ \sigma\ \mathrm{Root}_{x_i,k}(f)$, where $\mathrm{Root}_{x_i,k}(f)$, $1 \leq i \leq n$, is a real algebraic function (defined on some set), and $\sigma \in \{=, \neq, >, <, \geq, \leq\}$. Given an ETF Φ, we can always write it in a disjunctive normal form $\Phi = \vee_{i=1}^{s} \wedge_{j=1}^{s_i} \phi_{i,j}$. Let $\Phi_i := \wedge_{j=1}^{s_i} \phi_{i,j}$. Assume that the variables $x_1, \ldots, x_n$ are ordered as $x_1 < \cdots < x_n$. Let $v(\phi_{i,j})$ be the biggest variable appearing in $\phi_{i,j}$. Then we can always arrange the order of atomic formulas appearing in each Φ_i such that for any ϕ_{i,j_1} and ϕ_{i,j_2}, where $j_1 < j_2$, we have $v(\phi_{i,j_1}) \leq v(\phi_{i,j_2})$. Let $w \in \{x_1, \ldots, x_n\}$. Denote by $\Phi_i^{<w} := \wedge_{v(\phi_{i,j}) < w} \phi_{i,j}$. We say Φ_i is *proper* if for any $j = 1, \ldots, s_i$, if $\phi_{i,j} = v\ \sigma\ \mathrm{Root}_{v,k}(f)$, then $\mathrm{Root}_{v,k}(f(\alpha))$ is defined for all α satisfying $\Phi_i^{<v}$. We say Φ is proper if every Φ_i is proper. It is clear that a CAF is a proper restricted ETF.

Generalized Cylindrical Algebraic Formula (GCAF). Let S be a set of disjoint cells in a CAD of $\mathbb{R}^n$. A GCAF associated with S, denoted by Φ, is a proper restricted ETF $\Phi = \vee_{i=1}^{s} \wedge_{j=1}^{s_i} \phi_{i,j}$ such that

– the zero set of Φ is exactly $\cup_{c\in S} c$,
– the zero set of $\Phi_i := \wedge_{j=1}^{s_i}\phi_{i,j}$ is a union of some cells in S,
– the zero sets of Φ_i and Φ_j are disjoint for $1 \le i < j \le s$,
– each $\phi_{i,j}$ is of the form $v = \mathrm{Root}_{v,k}(f)$, where $v \in \{x_1, \ldots, x_n\}$,
– for every $w \in \{x_1, \ldots, x_n\}$, we have $\pi_{<w}(\Phi_i^{\le w}) = (\Phi_i^{<w})$, where $\Phi_i^{\le w} := \wedge_{v(\phi_{i,j})\le w}\phi_{i,j}$ and $\Phi_i^{<w} := \wedge_{v(\phi_{i,j})<w}\phi_{i,j}$.

A GCAF clearly has a cylindrical structure justifying its name. Note that both a CAF and a GCAF can be naturally encoded in a tree data structure, with each node representing an atomic formula $\phi_{i,j}$. This tree together with the CAD cells information is used in algorithms for simplifying CAFs in Section 4.

Example 2 *Both* $(-1 \le x \wedge x \le 1 \wedge -\sqrt{1-x^2} \le y \wedge y \le \sqrt{1-x^2})$ *and*

$$\begin{aligned}(x = -1 \wedge y = 0) \vee (-1 < x \wedge x < 1 \wedge -\sqrt{1-x^2} \le y \wedge y \le \sqrt{1-x^2})\\ \vee (x = 1 \wedge y = 0)\end{aligned}$$

are GCAFs equivalent to the CAF in Example 1.

The simplification procedure presented in this paper turns a CAF ϕ (in disjunctive normal form) into an equivalent GCAF Φ (in disjunctive normal form). We say Φ is simpler than ϕ if the number of conjunctive clauses in Φ is strictly less than that of ϕ.

3 Motivating examples

In this section, we go through several examples, coming from the application of automatic loop transformation, so as to introduce the heuristic strategies for combing CAFs. Those strategies are formally presented in Section 4. We refer the reader to [12, 7] for the application background of these examples.

Example 3 We consider polynomial multiplication with synchronous scheduling. It is formulated as the following quantifier elimination problem. If no simplification is applied, the result consists of 10 conjunctive clauses, as shown below.

```
ff := &E([i,j]), (0 <= i) &and (i <= n) &and (0 <= j) &and
                  (j <= n) &and (t = n - j) &and  (p = i + j);
R :=  PolynomialRing([i,j,p,t,n]);
sols := QuantifierElimination(ff, R, output=rootof, simplification=false);
‘&or‘(‘&and‘(n = 0,t = n,p = 0),‘&and‘(0 < n,t = 0,p = n),
‘&and‘(0 < n,t = 0,n < p,p < 2*n),‘&and‘(0 < n,t = 0,p = 2*n),
‘&and‘(0 < n,0 < t,t < n,p = -t+n),‘&and‘(0 < n,0 < t,t < n,-t+n < p,p < 2*n-t),
‘&and‘(0 < n,0 < t,t < n,p = 2*n-t),‘&and‘(0 < n,t = n,p = 0),
‘&and‘(0 < n,t = n,0 < p,p < n),‘&and‘(0 < n,t = n,p = n))
```

We observe that some conjunctive clauses can be merged into one. For instance, consider the subformula

$$\begin{aligned}(0 < n \wedge t = 0 \wedge p = n) \vee (0 < n \wedge t = 0 \wedge n < p \wedge p < 2n)\\ \vee (0 < n \wedge t = 0 \wedge p = 2n)\end{aligned}. \qquad (1)$$

Note that $(0 < n \wedge t = 0)$ is common to all three conjunctive clauses. Applying the distributivity law, the above formula is equivalent to

$$(0 < n \wedge t = 0) \wedge ((p = n) \vee (n < p \wedge p < 2n) \vee (p = 2n)) .$$

Observe that $(p = n) \vee (n < p \wedge p < 2n) \vee (p = 2n)$ can be combined into one conjunctive clause, namely $n \leq p \wedge p \leq 2n$. Thus, the above sub-formula of Equation (1) is equivalent to

$$0 < n \wedge t = 0 \wedge n \leq p \wedge p \leq 2n.$$

The above transformation can be explained in the language of CAD. Here $(0 < n \wedge t = 0)$ represents a CAD cell of $\mathbb{R}^2$ while $p = n$, $n < p \wedge p < 2n$, and $p = 2n$ represent respectively three adjacent children of it, which makes the combination straightforward. This observation forms our first idea. Applying this strategy to the entire expression in Equation (1) yields the following simplified formula:

```
‘&or‘(‘&and‘(n = 0,t = n,p = 0),
      ‘&and‘(0 < n,t = 0,n <= p,p <= 2*n),
      ‘&and‘(0 < n,0 < t,t < n,-t+n <= p,p <= 2*n-t),
      ‘&and‘(0 < n,t = n,0 <= p,p <= n)
```

Let us look at the last three conjunctive clauses. At first glance, it seems that they cannot be combined into one. A key observation is that if we specialize $-t + n \leq p \wedge p \leq 2n - t$ at $t = 0$ and $t = n$, we obtain $n \leq p \wedge p \leq 2n$ and $0 \leq p \wedge p \leq n$. Thus the last three conjunctive clauses can be combined into one:

```
‘&and‘(0 < n,0 <= t,t <= n,-t+n <= p,p <= 2*n-t).
```

Applying this *specialization technique* again, we obtain the final output:

```
‘&and‘(0 <= n,0 <= t,t <= n,-t+n <= p,p <= 2*n-t).
```

Combining together the two simplification techniques introduced in Example 3 forms the basis for the first level of our simplification procedure. Here is the result produced by the `QuantifierElimination` command of the `RegularChains` library at level `L1`:

```
QuantifierElimination(ff, R, output=rootof, simplification='L1');
((((0 <= n) &and (0 <= t)) &and (t <= n)) &and
(-t + n <= p)) &and (p <= 2 n - t)
```

Example 4 In this example, we consider polynomial multiplication with asynchronous scheduling. The following shows the output without using simplification. The output has 12 conjunctive clauses.

```
ff := &E([i,j]), (0 <= i) &and (i <= n) &and (0 <= j) &and (j <= n) &and
                 (t = n - j) &and  (p = i + j);
R :=  PolynomialRing([i,j,t,p,n]);
sols := QuantifierElimination(ff, R, output=rootof, simplification=false);
‘&or‘(‘&and‘(n = 0,p = 0,t = n), ‘&and‘(0 < n,p = 0,t = n),
```

```
‘&and‘(0 < n,0 < p,p < n,t = -p+n),
‘&and‘(0 < n,0 < p,p < n,-p+n < t,t < n),‘&and‘(0 < n,0 < p,p < n,t = n),
‘&and‘(0 < n,p = n,t = 0),‘&and‘(0 < n,p = n,0 < t,t < n),
‘&and‘(0 < n,p = n,t = n),‘&and‘(0 < n,n < p,p < 2*n,t = 0),
‘&and‘(0 < n,n < p,p < 2*n,0 < t,t < -p+2*n),
‘&and‘(0 < n,n < p,p < 2*n,t = -p+2*n),‘&and‘(0 < n,p = 2*n,t = 0))
```

Here is the result produced by the `QuantifierElimination` command of the `RegularChains` library with the simplification level set to `L1`:

```
((((n = 0) &and (p = 0)) &and (t = 0)) &or (((((0 < n) &and (0 <= p))
&and (p <= n)) &and (-p + n <= t)) &and (t <= n))) &or (((((0 < n)
&and (n < p)) &and (p <= 2 n)) &and (0 <= t)) &and (t <= -p + 2 n))
```

The situation here is a bit interesting. It seems that it is impossible to combine the three conjunctive clauses into one. This is because to make $-p+n \leq t \wedge t \leq n \Longleftrightarrow 0 \leq t \wedge t \leq -p+2n$ hold, we must have $-p+n=0$ and $-p+2n=n$, that is $n=p$ must hold. This obviously does not always hold under either the condition $0<n \wedge 0 \leq p \wedge p \leq n$ or the condition $0<n \wedge n<p \wedge p \leq 2n$.

However, if we look more closely at the example, we find that the formulas

$$\forall n,p,t \;\; 0<n \wedge 0 \leq p \wedge p \leq n \wedge -p+n \leq t \wedge t \leq n \implies 0 \leq t \wedge t \leq -p+2n.$$

and

$$\forall n,p,t \;\; 0<n \wedge n<p \wedge p \leq 2n \wedge 0 \leq t \wedge t \leq -p+2n \implies -p+n \leq t \wedge t \leq n.$$

are always true. Thus the last two can be combined into one

$$0<n \wedge 0 \leq p \wedge p \leq 2n \wedge -p+n \leq t \wedge t \leq n \wedge 0 \leq t \wedge t \leq -p+2n.$$

This third simplification technique forms the foundation of option 3 of the algorithm presented in the next section. Now, the simplified output consists of the following single conjunction: The following is the simplified output.

```
((((((0 <= n) &and (0 <= p)) &and (p <= 2 n)) &and (n - p <= t))
&and (t <= n)) &and (0 <= t)) &and (t <= 2 n - p)
```

Example 5 Consider a more advanced example from [7]. Without simplification, the output cannot be displayed completely. Below, we only display the first 2 and the last 2 conjunctive clauses. There are 223 conjunctive clauses, in total.

```
R := PolynomialRing([i, j, t, p, u, b, B, n]);
ff := &E([i,j]), (0 < n) &and (0 <= i) &and (i <= n) &and (0 <= j) &and
                 (j <= n) &and  (t = n - j) &and  (p = i + j) &and
                 (b>=0) &and (0<=u) &and (u<B) &and (p=b*B+u);
QuantifierElimination(ff,R,partial=true,output=rootof,simplification=false);
‘&or‘(‘&and‘(0 < n,0 < B,B < n,b = 0,u = 0,p = b*B+u,t = n),
‘&and‘(0 < n,0 < B,B < n,b = 0,0 < u,u < B,p = b*B+u,t = -u+n),
...
‘&and‘(0 < n,2*n < B,n/B < b,b < 2*n/B,u = -B*b+2*n,p = b*B+u,t = 0),
‘&and‘(0 < n,2*n < B,b = 2*n/B,u = 0,p = b*B+u,t = 0))
```

If the first two simplification techniques (options 1 or 2 in Section 4) are used, there are 29 conjunctive clauses. We only display the first 2 and the last 2 ones.

```
‘&or‘(
‘&and‘(0 < n, 0 < B, B < n, 0 <= b, b <= -(B-n)/B, 0 <= u, u < B,
       p = b*B+u, -B*b+n-u <= t, t <= n),
‘&and‘(0 < n, 0 < B, B < n, -(B-n)/B < b, b < n/B, 0 <= u, u <= -B*b+n,
       p = b*B+u, -B*b+n-u <= t, t <= n),
...
‘&and‘(0 < n, 2*n < B, 0 < b, b < n/B, -B*b+n < u, u <= -B*b+2*n,
       p = b*B+u, 0 <= t, t <= -B*b+2*n-u),
‘&and‘(0 < n, 2*n < B, n/B <= b, b <= 2*n/B, 0 <= u, u <= -B*b+2*n,
       p = b*B+u, 0 <= t, t <= -B*b+2*n-u))
```

If we use Option 3, the output consist only 5 conjunctive clauses:

```
‘&or‘(
‘&and‘(0 < n,0 < B,B < 2*n,0 <= b,b <= 2*n/B,0 <= u,u < B,u <= -B*b+2*n,
       p = B*b+u, -B*b+n-u <= t, t <= n, 0 <= t, t <= -B*b+2*n-u),
‘&and‘(0 < n, B = 2*n, b = 0, 0 <= u, u < 2*n,
       p = u, n-u <= t, t <= n, 0 <= t, t <= 2*n-u),
‘&and‘(0 < n, B = 2*n, 0 < b, b < 1/2, 0 <= u, u <= -2*b*n+2*n,
       p = 2*b*n+u, -2*b*n+n-u <= t, t <= n, 0 <= t, t <= -2*b*n+2*n-u),
‘&and‘(0 < n, B = 2*n, 1/2 <= b, b <= 1, 0 <= u, u <= -2*b*n+2*n,
       p = 2*b*n+u, 0 <= t, t <= -2*b*n+2*n-u),
‘&and‘(0 < n, 2*n < B, 0 <= b, b <= 2*n/B, 0 <= u, u <= -B*b+2*n,
       p = B*b+u, -B*b+n-u <= t, t <= n, 0 <= t, t <= -B*b+2*n-u))
```

The situation here is more subtle. It can be shown that the third one and the fourth one can be combined together using the technique shown in option 3. But it can not be combined with the second one by specialization. This is because in the second one we have $0 \leq u \wedge u < 2n$ while in the third one we have $0 \leq u \wedge u \leq -2bn + 2n$. When $b = 0$, the latter is equivalent to $0 \leq u \wedge u \leq 2n$. So a more general “pivot formula” is needed, which can be found in the first and the fifth conjunctive clauses. The technical details are covered in Section 4. This idea form the basis of option 4, which now brings a single conjunctive clause.

```
‘&and‘(0 < n,0 < B,0 <= b,b <= 2*n/B,0 <= u,u < B,u <= -B*b+2*n,
       p = B*b+u, -B*b+n-u <= t, t <= n, 0 <= t, t <= -B*b+2*n-u)
```

4 Algorithm

In this section, we present a heuristic algorithm Merge for combining cylindrical algebraic formulas, motivated by the examples shown in last section. In our algorithm, we have several levels of simplification. The most advanced level requires to decide the truth value of a quantifier free formula, which can be accomplished by a special QE procedure such as the one in [5] for computing triangular decomposition of semi-algebraic systems. In the following, we provide the pseudo

code for the algorithm and explain the subroutines in detail. The explanation supplies a loose proof of the correctness of the algorithms.

The function Merge takes a CAF tree T as input and returns an equivalent GCAF tree T. In below, we say a child node is *even* if it represents a section cell and *odd* if it represents a sector cell. Merge is called recursively to merge its odd children subtree first. The reason for doing this is that the representation of the odd nodes might be used as a reference to merge the even nodes. At last, it calls BasicMerge to merge the children of r, each of which is a rooted GCAF tree.

In BasicMerge, one type of subtree is treated in a special manner. It is a subtree rooted at a node c in T consisting of a single path. Let Γ be such a subtree of height h. It can be represented by a list of nodes $\{c_0, c_1, \ldots, c_h\}$, where $c_0 := c$ and c_i is the child of c_{i-1}, $i = 1, \ldots, h$. Each c_i is uniquely identified by an atomic formula $\psi(c_i)$ in a CAD tree data structure. In the following subroutines, we use $c.nodeRep$ to denote $\psi(c_0)$, $c.rep$ to denote $\wedge_{i=0}^{h}\psi(c_i)$, $c.desRep$ to denote $\wedge_{i=1}^{h}\psi(c_i)$, and $c.desDesRep$ to denote $\wedge_{i=2}^{h}\psi(c_i)$. Denote by $c.ancRep$ the conjunction of c's ascendants' representing atomic formulas.

In the context of the subroutines, these objects are all well defined. The top level function Merge always compresses such a tree Γ into a node. The procedure starts by dividing children of r into blocks s.t. children in different blocks are not adjacent. If opt is 1, the child node which has children is not processed. If opt is 3 or 4, the procedure tries to combine all the children nodes in a block into one. If it fails, opt 2 is adapted to combine as much adjacent children nodes in a block as possible. Note that we can also merge the nodes of the type $x_i < Root_{x_i,k}f$ and $x_i > Root_{x_i,k}f$ into one $x_i \neq Root_{x_i,k}f$ even the two nodes are not adjacent. This justifies the call of IneqMerge.

The procedure BlockMergable is used to test if the children of r in a block can be merged into one. Note that this function only handle the case that none of the children nodes in the block have children. To merge the children nodes which themselves have children, extra cost might be paid, since each subtree rooted at a child node is for sure not a tree consisting of a single path. The conjunction of the odd children is selected first as a pivot. If it fails and opt is 4, the conjunction of odd siblings of r is selected as a pivot.

When the procedure BlockMergable fails, the function NextMergable takes over to merge adjacent nodes in a block incrementally, which calls SameDesRep to test if the representation of the descendants of two adjacent nodes are equivalent. The equivalence is conducted by some simple test. In the case that the adjacent nodes have children, one checks if the subtrees rooted at them have physically the same representation by calling ExactSameDesRep. Otherwise, some simple specialization is conducted to check the equivalence.

5 Experimentation

In this section, we apply the algorithms of Section 4 to more examples. A comparison with the `Reduce` command of `Mathematica` is also provided.

Algorithm 1: Merge(r, T, opt)

Input: A CAF tree T rooted at r, an option 'opt' on simplification level.
Output: An equivalent GCAF tree T.

```
1 begin
2   if r has no children then return ;
    // The recursive call is first made for the odd children
3   for each odd child c of r do Merge(c, T, opt) ;
4   for each even child c of r do Merge(c, T, opt) ;
5   BasicMerge(r, T, opt);
```

Example 6 [11] This example is about scanning index sets generated by applying a non-linear schedule. Without simplification, the output has 58 clauses. With option 1, the output has 10 conjunctive clauses. With option 3, the output has exactly one conjunctive clause, although it takes about 200 more seconds.

```
ff := (n>=7) &and (2<=x) &and (x<=n) &and (4<=y) &and (y<=n)
             &and (n-x<=y) &and (t=(n-3)*x+y);
R := PolynomialRing([y, x, t, n]);
QuantifierElimination(ff,R,output=rootof,simplification='L3');
'&and'(7 <= n,3*n-8 <= t,t <= n^2-2*n,2 <= x,x <= -(n-t)/(n-4),
       -(n-t)/(n-3) <= x,x <= (t-4)/(n-3),x <= n,y = -n*x+t+3*x)
```

Example 7 [11] This example is about scanning index sets generated by normalizing loop strides. Without simplification, the output has 8 conjunctive clauses. The option 1 is sufficient to simplify it into one piece.

```
ff := (i>=2) &and (i^2<=n) &and (k>=i) &and (k*i<=n) &and (j=k*i);
R := PolynomialRing([k, i, j, n]);
QuantifierElimination(ff,R,output=rootof,simplification='L1');
'&and'(4 <= n,4 <= j,j <= n,2 <= i,i <= j^(1/2),k = j/i)
```

Example 8 [2] An example illustrating simple CAD. Without simplification, there are 51 clauses. After simplification with option 1, we have

```
R := PolynomialRing([z, y, x]);
qff := &E([z]), (19*z - 10*x + 10*y < 0) &and ((x^2+y^2+(z-3)^2<9)
                &or (2*x+19*z+10*y-11>=0));
QuantifierElimination(qff, R, output=rootof, simplification='L1');
'&or'('&and'(190/187-10/187*922^(1/2) < x, x < 11/12,
             100/461*x-570/461-19/461*(-561*x^2+1140*x+900)^(1/2) < y,
             y < 100/461*x-570/461+19/461*(-561*x^2+1140*x+900)^(1/2)),
      '&and'(x = 11/12,-1435/1383-19/5532*212199^(1/2) < y,
             y < -1435/1383+19/5532*212199^(1/2)), 11/12 < x)
```

Example 9 An example from program termination analysis. Without simplification, the output has 246 conjunctive clauses. After simplification with option 2, the output has 14 conjunctive clauses.

Algorithm 2: BasicMerge(r, T, opt)

Input: A GCAF tree T rooted at r, an option 'opt' on simplification level.
Output: A simplified GCAF tree T.

```
1  begin
2      if r has no children then
3          return;
4      else if r has only one child then
5          let c be the only child of r;
6          if c has children then return;
7          r.rep := r.nodeRep ∧ c.rep; r.children := ∅, return;
8      put the adjacent children of r into the same block;
9      if opt = 1 then put each child having children in a separate block;
10     NCL := ∅; // NCL means new children list
11     for each block B do
12         if opt = 3 or opt = 4 then
13             bool, pivotRep := BlockMergable(B, opt);
14             if bool then
15                 key := BlockMerge(B, pivotRep);
16                 NCL.append(key); next;
17         MGL := {B[1]};// MGL means a merge list
18         pivot := −1; m := |B|;
19         for i from 1 to m do
20             if NextMergable(B, i, m, pivot, opt) then
21                 MGL := MGL.append(B[i + 1]);
22             else
23                 if |MGL| = 1 then
                       // no new children created
24                     NCL.append(B[i]);
25                 else
26                     key := BlockMerge(MGL, B[pivot].rep);
27                     NCL.append(key);
28                 if i < m then MGL := {B[i + 1]} ;
29     if opt ≠ 1 then
30         if |NCL| = 2 or |NCL| = 3 then
31             NCL := IneqMerge(NCL);
32     if |NCL| = 1 then
33         let c be the only element of NCL;
34         if c has children then
35             r.children := NCL;
36         else
37             r.rep := r.nodeRep ∧ c.rep; r.children := ∅;
38     else
39         r.children := NCL;
```

Algorithm 3: BlockMergable(B, T, opt)

Input: A block of adjacent children in one stack.
Output: If the children can be combined into one by the heuristic strategy below, return true and the combined representation; otherwise return false and empty.

```
begin
    if at least one node in B has children then return false, ∅ ;
    let L be the odd nodes of B; let pivotRep := ∧_{b∈L} b.desRep; res := true;
    for b ∈ B do
        set res to the truth value of ∀x, (b.ancRep ∧ b.nodeRep ∧ b.desRep ⇔
        b.ancRep ∧ b.nodeRep ∧ pivotRep);
        if not res then break ;
    if res then return res, pivotRep ;
    if opt = 4 then
        let p be the parent of nodes in B;
        if p is not root and p is even node then
            let sL be the odd siblings of p;
            if |sL| ≠ 0 and none of sL has children then
                pivotRep := ∧_{s∈sL} s.desDesRep; res := true;
                for b ∈ B do
                    res := ∀x, (b.ancRep ∧ b.nodeRep ∧ b.desRep ⇔
                    b.ancRep ∧ b.nodeRep ∧ pivotRep);
                    if not res then break ;
                if res then return res, pivotRep ;
    return false, ∅
```

Algorithm 4: BlockMerge(B, pivotRep)

Input: A block B of adjacent nodes in a stack. A pivot representation pivotRep.
Output: A conjunctive ETF clause equivalent to $(\vee_{c\in B} c) \wedge pivotRep$.

```
begin
    create a new node c;
    let a and b be respectively the first and the last element of B;
    // Next we abuse the notation of intervals to represent a node
       for simplicity.
    if a = (a_1, a_2) then
        if b = (b_1, b_2) then r := (a_1, b_2) ;
        else if b = [b_1, b_1] then r := (a_1, b_1] ;
    else if a = [a_1, a_1] then
        if b = (b_1, b_2) then r := [a_1, b_2) ;
        else if b = [b_1, b_1] then r := [a_1, b_1] ;
    let c := r ∧ pivotRep;
    return c
```

Algorithm 5: NextMergable(A, i, m, pivot, opt)

Input: A block of adjacent nodes A, current node $A[i]$, final node $A[m]$ and pivot node *pivot*.
Output: Determine if $A[i]$ can be combined with its right adjacent sibling.
begin
  if $i = m$ **then**
    return false;
  if $pivot = -1$ **then**
    if $A[i]$ *is even node* **then**
      $pivot := i + 1$;
      if $SameDesRep(A[pivot], A[i], opt)$ **then** return true ;
      else return false; ;
    else
      $pivot = i$;
      if $SameDesRep(A[pivot], A[i + 1]), opt$ **then** return true ;
      else $pivot := -1$; return false; ;
  else
    if $A[i]$ *is even node* **then**
      if $SameDesRep(A[pivot], A[i + 1], opt)$ **then** return true ;
      else $pivot := i + 1$; return false; ;
    else
      if $SameDesRep(A[pivot], A[i + 1], opt)$ **then** return true ;
      else $pivot := -1$; return false; ;

Algorithm 6: SameDesRep(pivot, i, opt)

Input: A pivot node *pivot* and a node i in the same block.
Output: Test if the representation of their descendants are the same by some simple heuristics.
begin
  if *opt is* $2, 3, 4$ **then**
    if *both pivot and i have children* **then**
      return ExactSameDesRep(pivot, i);
  if *i is odd* **then** return the truth value of $i.desRep = pivot.desRep$;
  else return the truth value of $i.desRep = subs(i.nodeRep, pivot.desRep)$;

```
R := PolynomialRing([v1,v2,v3,labda,a11,a21,a22,a33,b12,b22,b23]);
f:= &E([v1,v2,v3,labda]), (labda>0) &and (a11*v1=labda*v1) &and
                    (a21*v1+a22*v2=labda*v2) &and (a33*v3=labda*v3) &and
                    (b12*v2>0) &and (b22*v2+b23*v3>0);
QuantifierElimination(f, R, output=rootof,partial=true,simplification='L2');
'&or'('&and'(b23 <> 0, b22 < 0, b12 < 0, a22 <= 0, a21 <> 0, 0 < a11),
'&and'(b23 <> 0, b22 < 0, b12 < 0, 0 < a22),
'&and'(b23 <> 0, b22 < 0, 0 < b12, 0 < a33, a22 <> a33, a21 <> 0, a11 = a33),
```

Algorithm 7: IneqMerge(L, opt)

Input: A list L of nodes in a stack.
Output: Return a list NL of merged nodes.
begin
  if $|L| = 2$ **then** $a := L[1]$; $b := L[2]$;
  else if $|L| = 3$ **then** $a := L[1]$; $b := L[3]$;
  else return L ;
  if $a.nodeRep = (-\infty, \alpha)$ *and* $b.nodeRep = (\alpha, +\infty)$ *and* $SameDesRep(a, b, opt)$ **then**
    create a new node c; $c.rep := x \neq \alpha \wedge a.desRep$;
    if $|L| = 2$ **then** return $[c]$;
    else if $|L| = 3$ **then** return $[c, L[2]]$;

```
'&and'(b23 <> 0, b22 < 0, 0 < b12, 0 < a33, a22 = a33),
'&and'(b23 <> 0, b22 = 0, b12 <> 0, 0 < a33, a22 <> a33, a21 <> 0, a11 = a33),
'&and'(b23 <> 0, b22 = 0, b12 <> 0, 0 < a33, a22 = a33),
'&and'(b23 <> 0, 0 < b22, b12 < 0, 0 < a33, a22 <> a33, a21 <> 0, a11 = a33),
'&and'(b23 <> 0, 0 < b22, b12 < 0, 0 < a33, a22 = a33),
'&and'(b23 <> 0, 0 < b22, 0 < b12, a22 <= 0, a21 <> 0, 0 < a11),
'&and'(b23 <> 0, 0 < b22, 0 < b12, 0 < a22),
'&and'(b23 = 0, b22 < 0, b12 < 0, a22 <= 0, a21 <> 0, 0 < a11),
'&and'(b23 = 0, b22 < 0, b12 < 0, 0 < a22),
'&and'(b23 = 0, 0 < b22, 0 < b12, a22 <= 0, a21 <> 0, 0 < a11),
'&and'(b23 = 0, 0 < b22, 0 < b12, 0 < a22))
```

Example 10 Consider computing a CAF of $p_1 \leq 0 \wedge p_2 \leq 0$, where p_1, p_2 are random polynomials of given degree d and with n variables. We have tested the case for $n = 2, d = 2, 3, 4$ and $n = 3, d = 2, 3$ and $n = 4, d = 2$. The experimentation shows that the number of conjunctive clauses after simplification using option 2 is about 3 to 5 times less than without applying simplification.

We have also selected some examples from [16], generated automatically from QEPCADexamplebank_v4.txt, to test the simplification procedure. The experimental results are summarized in Table 6. The examples in previous sections are included. The QuantifierElimination command is called with options partial='true', output='rootof' and different simplification flavors. In the table, L0 corresponds to the option simplification='false'. Two columns are dedicated to each simplification option, as well as to the `Reduce` command of `Mathematica`: the left column contains the time spent on calling the QE procedure while the right column gives the number of conjunctive GCAF clauses in the output.

From the table, we derive the following observations. For all the examples, there is almost no simplification overhead for the levels L1 and L2. This is because almost no algebraic computations are needed for those two levels. There are only two examples, namely NL-Schedule and Poly-Mul-Tile, for which the simplification levels L3 and L4 incur significant overhead but also reduce significantly the amount of conjunctive clauses. For examples coming from the applica-

tion of loop transformation, the levels L3 and L4 are effective. But for examples from other application domains, the level L2 is the best to choose considering the computation overhead, which is the default option of our QE procedure. W.r.t. the `Reduce` command, our QE procedure can generate less number of conjunctive clauses for 16 out of the 25 examples. For the three examples for which our QE procedure generates more conjunctive clauses, we have checked that the reason is that our routine ExactSameDesRep only checks whether two subtrees are physically the same while the `Reduce` command is more aggressive. Finding cost-effective heuristic strategies to handle this problem is left for future work.

6 Conclusions

We have presented a multi-level heuristic algorithm for simplifying cylindrical algebraic formulas, motivated by applications like automatic loop transformation in computer programs. The experimentation shows that the method can reduce significantly the number of conjunctive clauses of a CAF. Nevertheless, more work is required to obtain more compact output in less computing time. In particular, the following related work needs to be investigated. In Chapter 8 of his thesis [2], Brown presented algorithms for constructing cylindrical formulas, which are proper restricted extended Tarski formulas having cylindrical properties, but not necessarily GCAFs. An essential idea in his approach is the concept of "polynomially compatible". But it is not always easy to determine if two sections in different stacks are polynomially compatible, especially for cylindrical algebraic decomposition computed by the partial CAD approach [9] or the regular chain approach [6]. Moreover, the simplification techniques of [2] cannot handle the case taken care of by options 3 and 4 of our method. Nevertheless, the ideas of "polynomially compatible" and "truth-boundary cells" deserve to be investigated further and related to our approach. The adjacency algorithm [1] might also help determining whether two intro-stack sections can be combined.
Acknowledgments. Supported by the NSFC (11301524,11471307,61202131).

References

1. D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition II: an adjacency algorithm for the plane. *SIAM J. Comput.*, 13(4):878–889, 1984.
2. C. W. Brown. *Solution Formula Construction for Truth Invariant CAD's*. PhD thesis, University of Delaware, 1999.
3. C. W. Brown. Fast simplifications for tarski formulas based on monomial inequalities. *Journal of Symbolic Computation*, 47(7):859 – 882, 2012.
4. C. W. Brown and A. Strzeboński. Black-box/white-box simplification and applications to quantifier elimination. In *Proc. of ISSAC '10*, pages 69–76, 2010.
5. C. Chen, J. H. Davenport, J. May, M. Moreno Maza, B. Xia, and R. Xiao. Triangular decomposition of semi-algebraic systems. In S.M. Watt, editor, *Proceedings ISSAC 2010*, pages 187–194, 2010.
6. C. Chen and M. Moreno Maza. An incremental algorithm for computing cylindrical algebraic decompositions. *Computer Mathematics: Proc. of ASCM '12*, pages 199–222, 2014.

Example	L0		L1		L2		L3		L4		Reduce	
ArcSin-B	0.899	10	1.192	8	0.986	6	1.004	6	0.983	6	0.096	12
A-Real-Implicitization	0.977	6	1.021	5	1.013	2	1.264	2	1.309	2	0.069	5
Collision-A-from-B-H	2.340	13	2.344	1	2.273	1	2.328	1	2.374	1	0.180	1
Cyclic-4	0.851	2	0.868	2	0.880	1	0.874	1	0.861	1	0.055	2
Edges-Square-Product	57.378	249	57.507	5	58.158	5	58.188	5	58.668	5	4.753	7
Ellipse-A-from-B-H	2.305	12	2.352	2	2.345	2	2.439	2	2.376	2	0.400	2
Intersection-dagger-B	1.797	113	1.837	84	1.810	78	2.160	78	2.145	78	0.289	104
Kahan-A	1.034	17	1.414	10	1.107	10	1.129	10	1.113	10	0.116	10
Kahan-B	1.056	13	1.135	9	1.152	9	1.133	9	1.136	9	0.145	9
Loop-Norm	1.418	8	1.399	1	1.414	1	1.402	1	1.441	1	0.059	2
Loop-Termination	44.383	246	43.795	186	44.128	14	44.215	14	44.547	14	0.161	58
NL-Schedule	2.915	58	3.039	10	3.018	10	222.654	1	236.145	1	0.073	12
Off-Center-Ellipse	1.847	4	1.917	2	1.893	2	1.923	2	1.914	2	0.129	2
Parametric-Parabola	0.839	7	0.871	5	0.874	4	1.057	4	1.110	4	0.075	5
Poly-Mul-ASyn	1.191	12	1.232	3	1.257	3	1.672	1	1.692	1	0.054	6
Poly-Mul-Syn	0.959	10	1.028	1	1.015	1	1.379	1	1.360	1	0.052	2
Poly-Mul-Tile	12.001	223	12.522	29	12.333	29	152.901	5	207.635	1	0.123	50
Positivity-of-Quartic	1.480	10	1.497	4	1.575	3	1.473	3	1.468	3	0.094	4
Putnum-Example	2.872	76	2.897	8	3.016	8	2.934	8	2.941	8	0.430	6
Random-dagger-A	2.106	214	2.150	160	2.162	148	2.209	148	2.146	148	0.155	189
Range-of-Lower-Bounds	1.057	2	1.121	1	1.119	1	1.091	1	1.157	1	0.075	1
Sphere-Half-Space	2.297	9	2.319	3	2.298	3	2.302	3	2.423	3	0.076	2
Whitney-Umbrella	0.908	8	0.964	7	1.003	3	1.257	3	1.455	3	0.066	7
X-axis-Ellipse-Problem	12.099	104	12.322	24	13.070	24	12.552	24	12.398	24	0.466	16
YangXia	2.225	8	2.245	2	2.257	2	2.897	2	2.850	2	0.101	2

Table 1. Comparison between different simplification options and `Reduce`.

7. C. Chen and M. Moreno Maza. Quantifier elimination by cylindrical algebraic decomposition based on regular chains. In *Proc. of ISSAC '14*, pages 91–98, 2014.
8. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Springer Lecture Notes in Computer Science*, 33:515–532, 1975.
9. G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 12(3):299–328, 1991.
10. A. Dolzmann and T. Sturm. Simplification of quantifier-free formulas over ordered fields. *Journal of Symbolic Computation*, 24:209–231, 1995.
11. A. Größlinger. Scanning index sets with polynomial bounds using cylindrical algebraic decomposition. Number MIP-0803. 2008.
12. A. Größlinger, M. Griebl, and C. Lengauer. Quantifier elimination in automatic loop parallelization. *J. Symb. Comput.*, 41(11):1206–1221, 2006.
13. H. Iwane, H. Higuchi, and H. Anai. An effective implementation of a special quantifier elimination for a sign definite condition by logical formula simplification. In *CASC*, volume 8136 of *LNCS*, pages 194–208. 2013.
14. A. Strzeboński. Computation with semialgebraic sets represented by cylindrical algebraic formulas. In *Proc. of ISSAC '10*, pages 61–68. ACM, 2010.
15. A. Strzeboński. Solving polynomial systems over semialgebraic sets represented by cylindrical algebraic formulas. In *Proc. of ISSAC '12*, pages 335–342. ACM, 2012.
16. D. J. Wilson. Real geometry and connectedness via triangular description: Cad example bank, 2013.