Nonlinear Polynomials, Interpolants and Invariant Generation for System Analysis*

—Preliminary Draft—

Deepak Kapur

Dept. of Computer Science,
University of New Mexico, USA,
kapur@cs.unm.edu

Abstract. System Invariant properties at various locations play a critical role in enhancing confidence in the reliability of system behavior, be it software, hardware and hybrid systems. While there has recently been considerable interest in researching heuristics for generating loops invariants, almost all developments have focused on generating invariants typically handled using SMT solvers including propositional formulas, difference and octagonal formulas and linear formulas. While we have been investigating methods based on symbolic computation algorithms including Gröbner basis and approximate quantifier elimination for over a decade (see [14, 36, 42, 45, 26, 27, 43, 44, 28, 50, 49, 16, 18] for some of our papers), the SMT and CAV community have only recently started considering nonlinear polynomial invariants since many programs, especially linear filters, hybrid systems, and other applications, need nonlinear invariants for analysis of their behavior.
We present an overview of our research with a focus on our work on nonlinear invariant generation [16] as well as interpolant generation [18] from the perspective of their role in software and hybrid system analysis. Our approach is in sharp contrast to some recent approaches in which nonlinear polynomials are approximated using linear inequalities and symbolic-numeric techniques. We also present new research on quantifier elimination heuristics for invariant generation and interpolant generation. Particularly, we give efficient algorithms for interpolant generation for quantifier-free theories of equality on uninterpreted symbols and octagonal formulas. We outline problems and challenges for future research.

1 Introduction

We give an overview of the elimination methods-both algebraic, geometric and logical, being pursued in our research group for static system analysis. We discuss various heuristics based on elimination for automatically generating loop

* This research has been partially supported by the NSF awards CCF-1248069 and DMS-1217054, and the CAS/SAFEA International Partnership Program for Creative Research Teams, as well as by Visiting Professorship for Senior International Scientists, the Chinese Academy of Sciences.

invariants, termination analysis, interpolant generation and related constructions found useful for static program analysis. The first part of the note provides an overview of ideal theoretic and quantifier elimination approaches for automatically generating polynomial equalities and inequalities as loop invariants for programs operating on numbers. Further, input-output specifications or postconditions of programs are not needed. This report borrows from discussions and examples from an earlier report in [28] which focussed on our work from until 2006. The second part discusses geometric and local heuristics for quantifier-elimination for generating octagonal invariants. This is followed by our research on quantifier elimination based approach for generating interpolants to derive efficient algorithms for subtheories. Although we have been investigating quantifier elimination based approach for interpolant generation for nearly a decade, the algorithms for interpolant generation for the theory of equality over uninterpreted symbols as well as octagonal formulas have never been presented before. Using a series of examples, it is demonstrated that even thought the worst case complexity of elimination methods is quite high, it is still possible to effectively use heuristics for elimination and get useful information even by hand. In this note, we have chosen to be informal by illustrating our techniques on simple examples; details and theoretical foundations can be found in our publications on these topics.

Let us begin with the following simple loop for computing the floor of the square root of a natural number.

Example 1. **function SquareRoot**(N: integer) **returns** a: integer
var a, s, t: integer **end var**
$a := 0, \quad s := 1, \quad t := 1;$
while $s \leq N$ **do**
$a := a + 1; \quad t := t + 2; \quad s := s + t;$
end while

Using the approach discussed in [42], a conjunction of two polynomial equations, $t = 2a + 1,\ s = (a + 1)^2$, can be automatically generated as an invariant. In fact, this formula can be shown to be the strongest invariant expressed in terms of polynomial equations. There is no need to provide a postcondition to drive invariants or give a priori shapes of the desired invariants. The second polynomial equation, $s = (a+1)^2$, though an invariant, is not an inductive invariant by itself; in other words, it is not the case that

$$s = (a + 1)^2 \quad \Longrightarrow \quad (((s = (a + 1)^2)|^s_{s+t})^t_{t+2})^a_{a+1}.$$ [1]

In contrast, each conjunct in an equivalent strongest formula $t = 2a + 1 \wedge\ s = -a^2 + at - a + t \wedge\ t^2 = 2a + 4s - 3t$ is an invariant as well as an inductive invariant. In a later paper [16], we gave a *saturation* based method for checking whether an assertion, such as $s = (a+1)^2$ for the above example, can be proved to be an invariant using *strengthening*.

[1] The notation $\alpha|^x_t$ stands for replacing all free occurrences of a variable x in a formula α by an expression t.

Here is a somewhat meaningless but historical example taken from a paper of Cousot and Halbwachs [10], where a method for generating linear inequalities as invariants was discussed using the abstract interpretation approach [8].

Example 2. **var** i, j: integer **end var**
$\langle i, j\rangle := \langle 2, 0\rangle$;
while b_1 **do**
if b_2 **then** $i := i + 4$;
else $i := i + 2, \quad j := j + 1$;
end if
end while

Polynomial methods discussed in [42, 45, 46] cannot be used to automatically generate an invariant for this example. However, methods based on quantifier-elimination [34] and Farkas' lemma [7] can generate an invariant for this example. Using the quantifier elimination approach illustrated in [34], the conjunction of inequalities $j \geq 0 \wedge i - 2j \geq 2$ can be easily deduced as an invariant.

Consider the following interesting example that we gave as a homework problem in a graduate course on semantics of programming languages at the University of New Mexico.

Example 3. **var** x, y, z: integer **end var**
$\langle x, y, z\rangle := \langle 1, a, 1\rangle$;
while $y > 0$ **do**
if $z = 0$ **then** $y := y - 1; \quad z := x$;
else $x := x + 1, \quad z := z - 1$;
end if
end while

The loop in this program does not admit any interesting polynomial invariant implying that the loop behavior cannot be captured polynomially. However, if a template involving program variables as well as their exponentiation is used, then using the same approach for elimination (really simplification), the following invariant can be generated: $(x + z) = 2^{(a+1)-y}$.

1.1 Organization

We first give a review of two radically different approaches based on elimination and symbolic computation algorithms. The first approach gives ideal theoretic semantics of program statements, where the ideal of invariants expressed as a conjunction of polynomial equalities is associated with every program location. The second approach assumes a priori that the shape of invariants of interest is known in the form of a template but exact invariants of that form must be derived. This approach uses quantifier elimination in the theory whose language is used to express invariants. Two subtheories considered are those of polynomial equalities as well as linear constraints. Polynomial inequalities can also be used but we have limited experience in considering them.

In 2009, we started investigating a geometric approach for approximating quantifier elimination for generating loop invariants for specialized theories. Inspired by Miné's work [38] on relational numerical constraints for specifying invariants in the ASTREE system [9], we decided to focus on this fragment of Presburger arithmetic [32, 29]. Considering geometry of octagonal formulas, we studied different types of transformations in a loop body which resulted in the consequent of a verification condition to be an octagonal formula as well as identify conditions under which the verification condition is valid in the presence of octagonal tests along a program path. We construct tables of such parameter constraints a priori under different transformation; it can be proved that all such parametric constraints are also octagonal. We show that this can be done in $O(n^3)$ where n is the number of program variables. This approach has been extended to nonconvex max-plus and min-plus constraints thus allowing disjunctive invariants. However, the tables of parametric constraints become complex and involve lots of disjunctions [32]. The approach however is promising because of its reliance on geometry of invariants. We believe that the approach can be extended to template polyhedra in which the coefficients of variables in linear constraints of a template polyhedra are fixed except for lower and upper bounds

While the material in the earlier sections is based on our already published papers, the material in the later sections has not been published elsewhere. Section 5 outlines an approach for generating ranking functions based on templates and quantifier-elimination to show termination of loops. In [31], we proposed a direct relationship between quantifier elimination and interpolant generation for first-order theories. Based on quantifier elimination heuristics, Section 6 presents efficient algorithms for interpolant generation for two quantifier-free subtheories: equality over uninterpreted symbols (EUF) and octagonal formulas. The paper concludes with some remarks on future research including challenges faced by SMT and symbolic computation researchers.

2 Ideal-Theoretic Approach to Invariant Generation

Loop invariants are the key ingredient of the axiomatic approach towards program semantics, also called the Floyd-Hoare inductive assertion approach. The concept of invariant has been used in abstract algebra, particularly algebraic geometry, for nearly 200 years. However, in computer science, the use of invariants first appeared in a paper by Hoare on a proof of the program FIND. Since then, the use of invariants is ubiquitous in understanding system behavior, be it of software, hardware or hybrid systems. Because of our interest in elimination methods in algebraic geometry since the mid 1980s, it was extremely gratifying to find a close connection between the concept of loop invariants with algebraic concepts of invariants in invariant theory, as the reader would observe from the discussion in this section.

For example 1 in the introduction, the values of program variables a, s, t after the $(i+1)^{th}$ iteration, written as a_{i+1}, s_{i+1}, and t_{i+1}, can be specified in terms

of their values in previous iterations as

$$a_{i+1} = a_i + 1, \quad s_{i+1} = s_i + t_i + 2, \quad t_{i+1} = t_i + 2,$$

with the initial values $a_0 = 0, s_0 = 1, t_0 = 1$; furthermore, $a_i = i$, the loop index. These recurrences can be solved in many cases in terms of the loop index. If the loop index can also be eliminated from these solutions, relations among program variables can be computed which do not depend upon the loop index. In this way, loop behavior can be characterized independently of the loop index.

In our 2003 paper [41], significant progress was reported in deriving loop invariants automatically using related ideas. Below, we discuss the key ideas and present results; more details can be found in [41, 42, 44].

It was proved in [42] that if one just considers polynomial equations (of any degree) as atomic formulas for specifying invariant properties of programs, then these polynomial invariants have a nice algebraic structure, called a *radical ideal* of polynomials. Given two invariants at a program location, expressed as polynomial equations $p_1 = 0$ and $p_2 = 0$, the following are also invariants:

1. $p_1 + p_2 = 0$,
2. $qp_1 = 0$ for any polynomial q, and
3. if $p_1 = {p_3}^k$ for some p_3 and $k > 0$, then $p_3 = 0$ is also an invariant.

The above are precisely the defining properties of a radical ideal of polynomials. In [42], this radical ideal associated with a program location was called the *polynomial invariant ideal*.

Theorem 1. *The set of invariants expressed as polynomial equations in $Q[x_1, \ldots, x_n]$ at a given program location constitute a radical ideal, called polynomial invariant ideal. Further, any elimination ideal of this radical ideal is also a polynomial invariant ideal.*[2]

By Hilbert's basis theorem, every ideal over a Noetherian ring has a finite basis. So a polynomial invariant ideal has a finite basis as well. From this finite basis, a formula which has the structure of a conjunction of disjunctions of polynomial equations can be generated, from which every polynomial invariant follows. Interestingly, disjunctive polynomial invariants can be easily expressed in the language using a polynomial equation since $pq = 0$ is equivalent to $p = 0 \vee q = 0$. Disjunctive invariants are usually not as easy to express in other frameworks, particularly those based on abstract interpretation. The Illinois cache coherence protocol problem discussed in [28] is an excellent illustration of the expressive power of conjunctions of disjunctions of polynomial equations as inductive assertions.

The problem of discovering invariants at a program location thus reduces to computing the associated polynomial invariant ideal at the location. If this cannot be achieved, our goal is to find the closest possible approximation to this

[2] Given an ideal I over a polynomial ring $Q[x_1, \ldots, x_n]$, its j-th elimination ideal I_j is the set of polynomials only in variables $x_{j+1}, \ldots x_n$ in I.

polynomial invariant ideal, which in ideal-theoretic terms, means computing a subideal, to this ideal, including the zero ideal, which corresponds to the formula `true`.

Assuming that a significant component of program states at a program location can be specified by a conjunction of disjunction of polynomial equations, our approach automatically derives loop invariants without any information about input-output specification and/or post condition of a program based on the following ideas:

1. Under certain conditions, semantics of programming language constructs can be given in terms of ideal-theoretic operations. A program becomes a computation on ideals. For programs manipulating numbers, a significant component of program states can be characterized using radical ideals.
2. The polynomial invariant ideal associated can be computed as a fixed point. The challenge is to determine conditions under which this fixed point computation terminates in a finite number of steps, or can be approximated so that the approximation can be computed in finitely many steps. Below such conditions are given by imposing restrictions on programs (see [42] for precise definitions).

The reader would observe that negation of polynomial equations is not allowed. It is an open problem how this approach can incorporate negated equations as a part of an invariant.[3]

Semantics as Ideal Operations In [42], we gave a procedure for computing the polynomial invariant ideal of a simple loop in a program. The semantics of programming language constructs is given in terms of manipulating radical ideals (equivalently, algebraic varieties) characterizing program states [45].

Similar to every Hoare triple $\{P\}\ \ S\ \ \{Q\}$ (assuming termination), an input radical ideal I characterizes (approximation of) states before the execution of S, and an output radical ideal J characterizes (approximation of) states after the execution of S. Thus P is a conjunction of polynomial equations corresponding to a finite basis of I and Q similarly corresponds to J. For the forward propagation semantics, the strongest possible postcondition Q for any given P, translates to generating maximal nontrivial radical ideal J from a given radical input ideal I. For the backward semantics, the weakest possible precondition P from any given Q, in ideal-theoretic terms, is equivalent to generating minimal nonzero

[3] A negated polynomial equation, say $p \neq 0$, can be expressed as a polynomial equation $pz = 1$, where z is a new variable using Rabinowitsch's trick (see [24]). It is unclear how such variables can be systematically introduced to generate invariants which have negated equations. A quantifier free formula that is a conjunction of polynomial equations and inequations defines a *quasi-variety*, which have been studied in automated geometry theorem proving [48]. It will be interesting to generalize the method of [42] to work on algebraic quasi-varieties instead of algebraic varieties.

radical ideal I from a given radical ideal J.[4] The initial ideal is determined by the input state.

For an assignment statement of the form $x := e$, the strongest postcondition corresponding to a precondition P is $\exists x'(x = e|^x_{x'} \wedge P|^x_{x'})$, whereas the strongest precondition corresponding to a postcondition Q is $Q|^x_e$. Thus $Q|^x_e$ is equivalent to substituting e for x in the ideal basis corresponding to Q and then recomputing its radical ideal. If the assignment is invertible, then strongest postcondition semantics is also relatively easy to compute by substituting for variables, otherwise a new variable x' must be introduced to stand for the previous values of x before the assignment, and the elimination ideal is computed after eliminating x' from the radical ideal corresponding to $P|^x_{x'}$ and the polynomial equation $x = e|^x_{x'}$.[5]

The semantics of a conditional statement is often approximated depending upon the condition in the statement. If a condition c is expressed as boolean combination of polynomial equations, then its effect can be expressed using ideal-theoretic operations. Otherwise, the condition c can be approximated by another condition d such that $c \implies d$ and d is a boolean combination of polynomial conditions. The coarsest approximation is where d is `true` (the corresponding ideal is the trivial zero ideal). Since merging of different control paths in a program (due to a conditional statement or a while statement) leads to the union of states corresponding to each path, this is represented logically as a disjunction of formulas corresponding to each path. In ideal-theoretic terms, it amounts to the intersection of the corresponding ideals. For example, if one path leads to $x = 0$ and the other path leads to $x = 1$, when these path merge, states are characterized by $x = 0 \vee x = 1$. For the first path, the ideal is $< x >$, and the ideal for the second path is $\langle x - 1 \rangle$. The intersection of these ideals is $\langle x(x-1) \rangle$, which captures the disjunction.

In case of a location where program control can pass arbitrarily many times, an approximation of the ideal corresponding to the states at that control point eventually stabilizes, i.e., the fixed point is reached approximating the polynomial invariant ideal.

2.1 Termination of Polynomial Invariant Ideal Computation

In [42, 44], we gave a procedure for computing an approximation of the polynomial invariant ideal. The most challenging part in this approach has been establishing termination of the procedure. Even though to date, there is no example for which the procedure does not terminate, establishing its termination in general is still an open problem. We have been able to show the termination of

[4] It might be useful to recall relationship between formulas and the associated ideals. If a formula $f \implies g$, then the ideal associated with g is a subideal of the ideal associated with f, since the set of satisfying assignments of f (zero set of polynomials) is contained in the set of satisfying assignments of g.

[5] This suggests that e appearing on the right side of an assignment can be an arbitrary polynomial.

the procedure only under certain technical conditions, particularly if assignment mappings are solvable with their eigenvalues as rational numbers (see [41] for precise details and proofs). This procedure uses a Gröbner basis algorithm for computing various ideal-theoretic operations.

The termination of the fixed point computation is established under these conditions by making use of a beautiful result in algebraic geometry that every algebraic variety can be decomposed into finitely many irreducible components [12]. This result is used to show that the algebraic variety of the states at the loop entry has irreducible components such that the dimension of at least one component goes up in every iteration of the procedure or the variety stabilizes, leading to the termination of the procedure. Since the dimension of a polynomial invariant ideal is bounded by the number of program variables, termination of the fixed point computation is guaranteed in steps bounded by the number of program variables in a program.

Invariants generated by this approach are strongest relative to these assumptions/approximations made in the semantics of programming constructions (e.g. tests in conditional statements and loops),

The following three results about the termination of the invariant ideal generation for programs with simple loops were proved in [42].

Theorem 2. *If the procedure for computing polynomial invariant ideals terminates, then it computes a polynomial invariant ideal of a given simple loop. Further, the approach of computing polynomial invariant ideals is complete and semidecidable for generating the strongest possible polynomial invariants of a simple loop.*

Theorem 3. *If sequences of assignment statements along different branches in a simple loop body can be executed in any order without affecting the semantics of the body, then the polynomial invariant ideal generation procedure terminates in as many iterations as the number of branches.*

The above implies that the termination is independent of the number of program variables changing in the loop. An immediate corollary of the above theorem is:

Corollary 1. *If a simple loop body is branch-free, i.e., it is a sequence of assignment statements, the procedure for computing its polynomial invariant ideal terminates in a fixed number of iterations.*

Theorem 4. *If sequences of assignments along different branches in a simple loop body do not commute*[6] *and assignments are solvable mappings with rational numbers as their eigenvalues, then the invariant ideal generation procedure terminates in at most $m+1$ iterations, where m is the number of program variables that change in the loop body.*

Further details can be found in our papers [42, 45, 43, 44]. A generalization of the above approach to handle nested loops and procedure calls needs further investigation. Perhaps summarizing semantics of individual loops and procedures as radical ideal transforms can be used to explore such an extension.

[6] (in other words, order in which branches are executed matters)

3 Quantifier-Elimination Approach

The approach discussed in this section assumes that the shape of possible invariant properties of programs of interest is known; an interested reader is referred to [26–28] for further details. Perhaps, the shape information can be determined from the postcondition associated with the program or by doing an a priori analysis of the program body or a combination, but so far little is known about how that can be effectively done. This requirement is similar in spirit to the design of an abstract domain required in the abstract interpretation framework introduced by Cousot [8]. It is assumed that a shape is a parameterized formula in some theories which can be very general (i.e., a polynomial equality of degree k) or very restricted (i.e., terms appearing from a fixed finite set).

In setting up the analysis, (i) hypothesizes parameterized assertions at appropriate control points in a program are associated, (ii) verification conditions from them are generated, (iii) quantifier elimination heuristics are employed to eliminate program variables from the verification conditions to generate constraints on parameters such that (iv) parameter values satisfying these constraints lead to valid verification conditions. The hypothesized assertions instantiated with these parameter values are then invariants. It will become evident that it is not necessary to do full quantifier-elimination; instead it suffices to generate a "sufficiently interesting" quantifier-free formula implied by $\forall \mathbf{X}\ \ \Gamma(\mathbf{P}, \mathbf{X})$, a verification condition generated using parameterized formulas, where $\mathbf{P}$ are the parameters and $\mathbf{X}$ are the program variables. Below, we review some results from [26, 27].

3.1 Expressing Shapes using Parameterized Polynomial Relations

Example 2 continued.: Let us assume that the quantifier-free theory of parameterized Presburger arithmetic is used for specifying invariants. I.e., an invariant $I(i, j)$ is hypothesized at the loop entry to be an inequality of the form $c_1 i + c_2 j + d \leq 0$, where c_1, c_2, d are unknown parameters. Values of c_1, c_2, d determine the shape of the invariant. That is, for example, if $c_1 = 0$, then i will not appear in the invariant.[7]

As in [10], it is assumed that the boolean test b_2 in the conditional statement is not an inequality; so it is abstracted to be `true`; if it was a linear inequality, it could have been used to further refine the verification conditions. Three verification conditions, two due to the branch in the loop body, and one from initialization, are:

$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0)$ (i),
$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0)$ (ii), and
$2c_1 + d \leq 0$. (iii)

Two problems to address are: (i) Does a given program location of interest satisfy a nontrivial invariant of the given shape? (ii) If it does, what is such an invariant, i.e., can parameter values be found so that when instantiated with these

[7] Of course, if $c_1 = 0, c_2 = 0, d = 0$, then the above formula simplifies to `true`, a trivial invariant.

specific values, the formula is indeed an invariant associated with the program location? Both of these questions are answered using quantifier-elimination by generating constraints on parameters from the verification conditions. To get the strongest possible invariant of the hypothesized form, a complete quantifier-elimination method is required. However, incomplete elimination heuristics can also be useful in deriving invariants.

For this example, for an invariant of the above shape to exist, each verification condition must be valid for all possible values of i, j.[8] To generate an invariant, values of c_1, c_2, d that make $\forall i, j, [(i) \wedge (ii)] \wedge (iii)$ valid, is computed. Had this formula not been valid, the invariant of the form $c_1 i + c_2 j + d \leq 0$ would not exist. Additional constraints on parmaters to rule out trivial invariants or satisfying certain conditions can be included as well. A quantifier-free formula implied by $\forall i, j, [(i) \wedge (ii)] \wedge (iii)]$ is:

$$\Psi = [(d \leq 0 \wedge c_1 = 0 \wedge c_2 \leq 0) \vee (2c_1 + d \leq 0 \wedge c_1 < 0 \wedge 2c_1 + c_2 \leq 0)].$$

For any values c_1, c_2, d that satisfy the above formula, $c_1 i + c_2 j + d \leq 0$ is an invariant. As an example, $c_1 = -2$, $c_2 = 0, d = 0$ satisfies the above constraints. Substituting for these values of parameters in the above template leads to $-2i \leq 0$ being an invariant. In fact, there are infinitely many values of c_1, c_2, d satisfying the above constraints.

It should be noted that these formulas fall outside the language of standard Presburger arithmetic. However, it is from the theory of parameterized Presburger arithmetic, in which coefficients of variables can be linear polynomials in parameters [34].

If an invariant for the above loop in Example 2 is hypothesized to be an nontrivial linear equation, the reader can verify that there does not exist such an nontrivial invariant (e.g., if $I(i, j) = (c_1 i + c_2 j + d = 0)$, since eliminating i, j, d from the verification condition results in $c_1 = 0, c_2 = 0, d = 0$, which simplifies I to the trivial invariant `true`).

To illustrate how nonlinear invariants can be generated using this approach as well, let us consider example 1 in the introduction for which there are multiple nonlinear invariants independent of each other.

`Example 1 continued:` Hypothesize an invariant of the loop to have a shape of a polynomial equation where the degree of each term is ≤ 2. That is,

$$I(a, s, t) \Leftrightarrow u_1\, a^2 + u_2\, s^2 + u_3\, t^2 + u_4\, as + u_5\, at + u_6\, st + u_7 a + u_8 s + u_9 t + u_{10} = 0,$$

where $u_1, \ldots, u_{10}$ are parameters. As discussed in detail in [34], constraints on parameters are generated using a heuristic for simplifying parametric polynomials (over the theory of an algebraically closed field), from which a basis for parameter values is generated, leading to multiple independent invariants. If there are multiple equations in the antecedent of a verification condition, then parametric Gröbner basis [25] constructions can be used for this purpose. In most cases, if there is a single equation in the antecedent, as is the case here, then

[8] Strictly speaking only values of i, j in reachable states.

it can be used directly to simplify by replacing the antecedent in the conclusion; even when there are many polynomial equations in the antecedent, they can be used as they are to simplify the conclusion without having to compute a parametric Gröbner basis.

Employing another heuristic that in a polynomial obtained after simplification, if the coefficient of each term in program variables is made 0, that suffices to make the polynomial to be 0 for all program variables, constraints on parameters can be generated. This is an incomplete but extremely useful strategy.

After eliminating program variables a, s, t from the verification condition, the following constraints on parameters are generated. Each of u_2, u_4, u_6 becomes 0, implying that the hypothesized shape of polynomial invariants can be further restricted by dropping out terms s^2, as, st. The following relations among other parameters are generated:

$$1.\ u_1 = -u_5,\ 2.\ u_7 = -2u_3 - u_5 + 2u_{10},\ 3.\ u_8 = -4u_3 - u_5,\ 4.\ u_9 = 3u_3 + u_5 - u_{10}.$$

The above set of constraints has infinitely many solutions. However, this infinite solution set can be finitely described. Each solution can be obtained from an independent set of 3 solutions obtained by making exactly one of the independent parameters, u_3, u_5 and u_{10}, to be 1, generating three invariants

$$t = 2a + 1,\ s = -a^2 + at - a + t,\ 4s = t^2 - 2a + 3t.$$

The reader would have noticed that these invariants are somewhat different from the ones given in the introduction. They are however logically equivalent. In fact, each of the above three invariants is also an inductive invariant. Whereas $s = (a+1)^2$ is a loop invariant, it is not an inductive invariant, as stated earlier. This invariant can be derived by combining $t = 2a + 1$ and $s = -a^2 + at - a + t$. Further, the first invariant is independent. The second invariant is independent of first one but not of the third one: It can be derived from the first one and third one. Similarly, the third invariant can be derived from the first and second. Even though these invariants were generated from independent solutions of a linear system of equations, variables standing for various power products in the linear system are related (particularly, a, at and t are related). To get a set of independent invariants, it thus becomes necessary to check derivability of one from the others.

The approach discussed in this section can be used to automatically generate invariants for programs with nested loops and procedure calls, as illustrated in [34]. The reader can consult [34] for details including many examples.

To generate a strongest possible invariant, there are two conditions which must be satisfied. Firstly, it should be possible for the subtheory from which parameterized formulas are drawn, to admit full quantifier-elimination. Secondly, in the generation of verification conditions, no approximation is made about the behavior of programming constructs. Both of these requirements can however be relaxed, but then the proposed approach may not generate the strongest possible invariant for a program. Further, even if the proposed approach declares that

there does not exist any nontrivial invariant of the hypothesized shape because of approximations made, an invariant of the hypothesized shape may still exist.

Neither of these two conditions were met for the above example. Even then the strongest invariant expressed as linear inequalities is generated. This suggests that these conditions do not always have to be satisfied to derive the strongest possible invariants.

It should also be noted that transfer functions needed in the abstract interpretation framework to express the semantics of program constructs on an abstract domain [11] can be generated using quantifier elimination; in fact quantifier elimination is the most general approach for computing transfer functions. We will not elaborate on this any further in the paper.

In a later section, it it shown how template based framework can be used to design ranking functions for proving termination of loops in a program using quantifier elimination.

4 Geometric Quantifier Elimination Heuristics for Octagonal Formulas

The high complexity of quantifier elimination algorithms as well as humungous output generated by them, if at all they output anything, can be daunting. It becomes thus critical to develop efficient heuristics for a subclass of formulas which have low complexity but more importantly, provide useful results, to make the proposed approach scalable. In this section, we will discuss practical heuristics for quantifier elimination for relational formulas using geometric techniques [32, 29]. Particularly, the sparse interaction among program variables occurring in verification conditions from octagonal formula and their special structure will allow for *localized reasoning*.

We have been successful in developing efficient polynomial time heuristics for a conjunction of formulas of the form $l \leq \pm x \pm y \leq h$ (also called *octagonal constraints* [38] or UTVPI constraints [22, 47]). These techniques are more likely to be scalable and thus useful for analysis of large programs Such numerical relational formulas have been found to be effective in ASTREE tool for its ability to detect bugs in large amounts of real code in flight control software, performing array bound checks, and finding memory leaks and other critical applications [9, 38]. Details of th heuristics can be found in [32]; below we sketch them.

An octagonal formula is of the form: $(l \leq \pm x \pm y \leq h)$ along with lower and/or upper bounds on variables. It is easy to visualize them as an octagon in two dimensions. Such constraints are simpler than general linear constraints Henceforth, by an atomic formula of the form $l \leq \pm x \pm y \leq h$, we mean any of the atomic formulas of the form $l \leq x + y \leq h, l \leq x - y \leq h, l \leq x \leq h, l \leq y \leq h$. By an octagonal expression, we mean any of the expressions in two distinct variables, say x, y for an instance: $x - y, x + y, -x - y, x, -x, A$, where A is an integer.

Octagonal formulas are also interesting to study from a complexity perspective and are a good compromise between interval constraints of the form $l \leq x \leq u$ and linear constraints. Linear constraint analysis over the rationals

($\mathbb{Q}$) and reals ($\mathcal{R}$), while of polynomial complexity, has been found in practice to be inefficient and slow, especially when the number of variables grows [38, 9], since it must be used repeatedly in an abstract interpretation framework. Often, we are interested in cases when program variables take integer values bound by computer arithmetic. If program variables are restricted to take integer values (which is especially the case for expressions serving as array indices and memory references), then octagonal constraints are the most expressive fragment of linear (Presburger) arithmetic over the integers with a polynomial time complexity. It is well-known that extending linear constraints to have three variables even with unit coefficients (i.e., ranging over $\{-1, 0, 1\}$) makes their satisfiability check over the integers to be NP-complete [22, 47]; similarly, restricting linear arithmetic constraints to be just over two variables, but allowing non-unit integer coefficients of the variables also leads to the satisfiability check over the integers being NP-complete.

4.1 A Geometric Local Heuristic

Given a program using n variables $x_1, \cdots x_n$, a parameterized formula of octagonal constraints expressing a program invariant at a given location is a conjunction of formulas of the form

$$octa(x_i, x_j) \triangleq l'_{i,j} \leq x_i - x_j \leq u'_{i,j} \wedge l_{i,j} \leq x_i + x_j \leq u_{i,j} \wedge l_i \leq x_i \leq u_i \wedge l_j \leq x_j \leq u_j,$$

for $i \neq j$, where $l_{i,j}, u_{i,j}, l'_{i,j}, u'_{i,j} l_j, u_j$ are parameters. A verification condition is:

$$\bigwedge_{1 \leq i \neq j \leq n} octa(x_i, x_j) \wedge C(X) \wedge C_k(X) \implies \bigwedge_{1 \leq i,j \leq n} octa(x'_i, x'_j), \qquad (1)$$

x'_i, x'_j are the new values of variables x_i and x_j after all the assignments along a k^{th} program branch in a loop body, $C(X)$ is a conjunction of all the loop tests on the k^{th} branch, and $C_k(X)$ is a conjunction of all the branch conditions along the k^{th} branch. There are no parameters appearing in $C(X), C^k(X), x'_i, x'_j$. The above verification condition has in general $2n * (n-1) + 2n = 2n^2$ parameters, which can be a big number for a large program with lots of variables The verification condition corresponding to all the branches in a loop body is then the conjunction of the verification conditions along each branch in the loop. In addition, the initial state of a program expressed by a precondition, as well as other initializing assignments to program variables the first time a loop body is entered, also impose additional constraints on parameters.[9]

To ensure that the above verification condition contains only octagonal formulas, all tests should be octagonal formulas and the assignment statements are of the form $x_i := x_i + A$, $x_i := -x_i + A$, $x_i := A$, for some constant integer A. Otherwise, tests and assignments must be approximated. In the worst case, an assignment will be approximated as x_i getting a random value. Similarly for a loop test that cannot be approximated, it is assumed to be true; for a conditional statement, it is assumed that both branches of the statement are executed. Some

[9] In case a program has too many branches in relation to its size, intermediate assertion can be used to decrease the number of branches that need to be analyzed.

preprocessing can be helpful to generate the desired verification conditions for programs which do not satisfy the above restrictions insofar as the simplified verification condition consists only of octagonal formulas; for example, a path can have a sequence of assignments which are more general than the above restricted assignments as long as the cumulative effect of all these assignments can be expressed satisfying the restrictions.

For generating invariants, the quantifier elimination problem to be solved is:

$$\forall X \bigwedge_{1\leq i\neq j\leq n} octa(x_i, x_j) \wedge C(X) \wedge C_k(X) \implies \bigwedge_{1\leq i,j\leq n} octa(x'_i, x'_j),$$

with the set of parameters P in the verification condition. It is also possible to include additional constraints on parameters in P such as certain parameters are nonzero. In general, octagonal expressions of program variables may not have any lower bound/upper bound, the domain on which parameters can take values are extended to include two constants $-\infty$ and $+\infty$ to stand, respectively, for no lower bound and no upper bound. Arithmetic operations and tests on the extended domain, which includes both $-\infty$ and $+\infty$, have to be appropriately extended to account for these values [35]. Unsatisfiable constraint solving becomes equivalent to some parameters taking $-\infty$ and $+\infty$ as their values, e.g, $u + 1 = u$ is satisfiable if u has $+\infty$ or $-\infty$ as its value.

If the above formula is satisfiable, this implies that there is indeed an invariant of the above form for the loop. A quantifier-free formula purely in parameters that is implied by the above verification condition is then generated. The approach is illustrated using a simple example; for additional examples, please consult [32].

Consider the following simple program.

Example 4. $x := 4;\ y := 6;$
while $y + x \geq 0$ **do**
if $y \geq 6$ **then** $x := -x;\ y := y - 1$ **else** $x := x - 1;\ y := -y;$
end while

Hypothesize an invariant at the loop entry of the form:

$$a \leq x \leq b \wedge c \leq y \leq d \wedge e \leq x - y \leq f \wedge g \leq x + y \leq h,$$

where a, b, c, d, e, f, g, h are parameters. The verification conditions resulting from the two branches are:

$(a \leq x \leq b \wedge c \leq y \leq d \wedge e \leq x - y \leq f \wedge g \leq x + y \leq h) \wedge (y + x \geq 0)) \implies$

$((y \geq 6 \implies (a \leq -x \leq b \wedge c \leq y-1 \leq d \wedge e \leq -x-y+1 \leq f \wedge g \leq -x+y-1 \leq h)) \wedge$

$(y \leq 5 \implies (a \leq x-1 \leq b \wedge c \leq -y \leq d \wedge e \leq x-1+y \leq f \wedge g \leq x-1-y \leq h)))$

We discuss below geometric heuristics for quantifier elimination based on the octagon corresponding to the hypothesis in the above verification condition gets affected by the assignment statements in each of the branches. The key idea is to find sufficient conditions on the parameters a, b, c, d, e, f, g, h for the octagon specified by the conclusion formula to include the octagon in the hypothesis

formula subject to the loop and branch test conditions. We have developed a case analysis based on how different kinds of assignments and various tests affect the validity of the verification condition leading to sufficient conditions on parameters. There is a table corresponding to each case of assignment statement, and an entry in the table corresponding to every atomic formula and test as discussed below. Such tables can be generated a priori once for all; details are given in [32].

General quantifier elimination tools are not likely to succeed, given that the complexity of generic quantifier elimination methods is exponential in the number of variables and alternations of quantifiers (in some cases, it is even worse, being of doubly exponential complexity). We have tried many of these examples on REDLOG,QPCAD, etc. but without much success.

4.2 Local Reasoning

A verification condition in general can be quite complex if relationship among all program variables is considered. However, in case of simple atomic formulas such as octagonal formulas, the verification condition can be considered *locally* by considering separately its subpart corresponding to each pair of distinct variables $x_i, x_j, i \neq j$. The results can be conjoined together accounting for limited interactions among variables in this subpart with other subparts; these details are explored in [32].

The subformula below corresponds to all the atomic formulas expressed only using x_i, x_j.

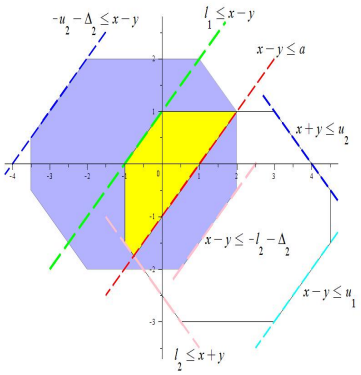
$$octa(x_i, x_j) \wedge C(X)_{[i,j]} \wedge C_k(X)_{[i,j]} \implies octa(x_i', x_j'), \tag{2}$$

By doing quantifier-elimination of program variables x_i, x_j on (2), generating sufficient conditions on the parameters in (2), and then taking a conjunction of such conditions on parameters for all possible pairs of variables, it is possible get a sufficient condition on all the parameters appearing in (1). This way, the analysis is localized to a single pair of variables, instead of having to consider all the variables together.

Consider a subformula of the above verification condition which relates a pair of distinct program variables x_i, x_j, expressed above as (2). To make the discussion less cluttered, we will replace x_i by x, x_j by y, as well as $l'_{i,j}, l_{i,j}, u'_{i,j}, u_{i,j}$, l_i, u_i, l_j, u_j by $l_1, l_2, u_1, u_2, l_3, u_3, l_4, u_4$, respectively; α stands for $C(X)_{[i,j]} \wedge C_k(X)_{[i,j]}$. To ensure that the verification condition has the form captured in (2) (particularly that the conclusion be $octa'(x, y)$), there are finitely many possibilities of the total cumulative effect on assignments for a distinct pair of variables x, y along a branch; each of these can be analyzed separately. All other cases must be approximated either by one of these assignments or by a random value.[10]

Possibility 1 $x := x + A$ and $y := y + B$;
Possibility 2 $x := -x + A$ and $y := -y + B$;

[10] In some cases, the cumulative effect of assignments of different forms may lead to one of the three possibilities above, in which case, they do not have to be approximated.



	present	absent
$x - y \leq a$	$a \leq u_1$ $a \leq -l_2 + \Delta_2$	$u_1 \leq -l_2 + \Delta_2$
$x - y \geq b$	$l_1 \leq b$ $-u_2 + \Delta_2 \leq b$	$-u_2 + \Delta_2 \leq l_1$
$x + y \leq c$	$c \leq u_2$ $c \leq -l_1 + \Delta_1$	$u_2 \leq -l_1 + \Delta_1$
$x + y \geq d$	$l_2 \leq d$ $-u_1 + \Delta_1 \leq d$	$-u_1 + \Delta_1 \leq l_2$
$x \leq e$	$e \leq u_3$ $e \leq -l_3 + A$	$u_3 \leq -l_3 + A$
$x \geq f$	$l_3 \leq f$ $-u_3 + A \leq f$	$-u_3 + A \leq l_3$
$y \leq g$ $B > 0$	$u_4 \geq g + B$	$u_4 = +\infty$
$y \geq h$ $B < 0$	$l_4 \leq h + B$	$l_4 = -\infty$

Fig. 1: Sign of only x is reversed in assignment.

Possibility 3 $x := -x + A$ and $y := y + B$.
Possibility 4 $x := A$ and $y := y + B$.
Possibility 5 $x := A$ and $y := -y + B$.
Possibility 6 $x := A$ and $y := B$.

Because of space limitations, we discuss below the third possibility corresponding to the above example. The table in Figure 1 corresponds to the third case, Other tables are included in [32].

The parametric verification condition in this third case is:

$$((l_1 \leq x - y \leq u_1 \wedge l_2 \leq x + y \leq u_2 \wedge l_3 \leq x \leq u_3 \wedge l_4 \leq y \leq u_4) \wedge \alpha) \Rightarrow$$
$$(-u_1 + \Delta_1 \leq x + y \leq -l_1 + \Delta_1 \wedge -u_2 + \Delta_2 \leq x - y \leq -l_2 + \Delta_2$$
$$\wedge \ -u_3 + A \leq x \leq -l_3 + A \wedge l_4 - B \leq y \leq u_4 - B),$$

where $\Delta_1 = A - B, \Delta_2 = A + B$, α is a conjunction of parameter-free atomic formulas from loop tests and branch conditions. These calculations have to be done once and for all and stored in a table. As an illustration, consider the case of how lower and upper bounds on $x - y$ are affected by the test $x - y \leq a$ in this third case. This is depicted in the Figure 1; the white octagon to the lower right side corresponds to the hypothesis octagonal constraints, the blue octagon is the result of assignments, with the red line corresponding to $x - y \leq a$. $(l_1 \leq x - y \leq u_1 \wedge x - y \leq a \wedge \gamma) \implies (l_2 \leq -x + A + y + B \leq u_2 \wedge \delta)$, where γ and δ are the remaining subformulas in the antecedent and conclusion, respectively, of the above verification condition in which atomic formulas expressing lower and upper bounds on $x - y$ do not appear. The entry $a \leq u_1 \wedge a \leq (-l_2 - \Delta_2)$ in the table in Figure 1 is the condition on u_1, l_2 for the above portion of the verification condition to be valid. If $x - y \geq b$ is also present, then the entry from the table gives constraints on l_1, u_2 to be $l_1 \leq b \wedge -u_2 - \Delta_2 \leq b$; if $x - y \geq b$ is absent instead, then the constraint on l_1, u_2 is $-u_2 - \Delta_2 \leq l_1$. There is an entry in the table for every possible atomic formula depending upon whether it is present or absent in α.

For the example discussed in the previous section, the constraint $a \leq 4 \leq b \wedge c \leq 6 \leq d \wedge e \leq -2 \leq f \wedge g \leq 10 \leq h$ is generated from the initial assignments to the variables. Using the tables in [32], constraints on the parameters are obtained for each branch. For the branch $x + y \geq 0 \wedge y \geq 6$, $A = 0, B = -1, \Delta_1 = 1, \Delta_2 = -1$, we get the entry from the table to be $g \leq 0 \wedge -f + 1 \leq 0$ and due to the absence of any upper bound on $y+x$, we get the entry $h \leq -e+1$. Since there is no condition on $x-y$, we get $f \leq -g-1$ and $-h-1 \leq e$; similarly, there is no condition on x, giving the constraint $a + b = 0$. However, y has a condition, namely $y \geq 6$ and $B < 0$, which gives $c \leq 5$; however, there is no upper bound condition on y, and since B is negative, no additional condition on parameters is imposed.

For the second branch corresponding to the condition $y + x \geq 0 \wedge \neg(y \geq 6)$ (which also imply $x \geq -5 \wedge y - x \leq 10$) [11], we similarly get from the table constraints $g \leq 0 \wedge e + 1 \leq 0 \wedge h \leq f + 1$ due to $y + x \geq 0$, and $5 \leq d \wedge -d \leq c \wedge 5 \leq -c$; in addition, we also get $a \leq -6$ due to $x \geq -5$ and $10 \leq -e \wedge -h-1 \leq -f \wedge 10 \leq -g - 1$ due to $y - x \leq 10$. Collecting all the constraints together:

$$(e \leq -10 \wedge f \geq 1 \wedge g \leq -11 \wedge h \geq 10 \wedge a \leq -6 \wedge b \geq 4 \wedge c \leq -5 \wedge d \geq 6) \wedge$$

$$(-1 \leq e + h \leq 1 \wedge g + f \leq -1 \wedge b + a = 0 \wedge -1 \leq h - f \leq 1 \wedge d + c \geq 0).$$

Values of a, b, c, d, e, f, g satisfying the above constraint result in an octagonal invariant for the loop in the above program, since the verification conditions generated from its two branches are valid for these values. By obtaining the maximum possible values for parameters standing for lower bounds and minimum possible values for parameters standing for upper bounds, the strongest possible invariant for the above program is generated. Making the lower bound parameters as large as possible, and the upper bound parameters as small as possible:

$$e = -10, f = 9, g = -11, h = 10, a = -6, b = 6, c = -5, d = 6.$$

The corresponding invariant is

$$-10 \leq x - y \leq 9 \wedge -11 \leq x + y \leq 10 \wedge -6 \leq x \leq 6 \wedge -5 \leq y \leq 6.$$

The correctness of the table entries (i.e., they generate correct parameter constraints in the sense that the parametric constraints after quantifier-elimination imply the table entries) can be verified. The reader would have noticed from the above examples as well as the tables that the constraints on parameters are also octagonal. In [32], a method for generating the strongest possible octagonal invariant from parametric constraints so generated is presented.

The following theorem is proved in [32].

Theorem 5. *Octagonal loop invariants of programs with simple loops (with no nesting of loops) can be automatically derived using the geometric quantifier elimination heuristics proposed above in $O(k * n^2)$ steps, where n is the number of program variables and k is the number of program paths.*

[11] These new conditions on the variables can be derived by local propagation.

We believe that similar analysis can be performed for a wider subclass of linear constraints. Particularly, we are investigating template polyhedra in which the linear expression is fixed (e.g., $2 * x - 3 * y + z$) with its lower and upper bounds being parameters.

5 Termination Analysis using templates, especially for generating nonlinear polynomial ranking functions

Template based approaches for generating ranking functions have been proposed in the literature. The approach we have been pursuing follows the same pattern as the one for loop invariant generation. We illustrate it using two examples:

Example 5. **var** x, y, z: integer **end var**
$\langle x, y, z\rangle$:=$\langle a, b, 0\rangle$;
while $y > 0$ **do**
if $y \ \ mod \ \ 2 = 0$ **then** $y := \frac{y}{2}$
else $y := \frac{y-1}{2}$; $z := z + x$;
end if
$x := 2x$;
end while

This example is quite trivial for showing termination since y is always decreasing. A termplate based approach will start with a linear template involving the program variables: $Ax+By+Cz+D$. Can we find parameter values such that the above polynomial strictly decreases over the integers in every iteration of the loop as well as it is always nonnegative (for well-foundedness of the ordering).

For it to be nonnegative always, initially it must be $Aa + Bb + D \geq 0$. In every iteration, for the branch when the test $y \ mod \ 2 = 0$ succeeds then $(Ax + By + Cz + D) > (2Ax + B\frac{y}{2} + Cz + D)$; for the second branch, $y \ mod \ 2 \neq 0] \implies (Ax + By + Cz + D) > (2Ax + B\frac{y-1}{2} + C(z + x) + D$. The constraint from the first branch simplifies to $(y > 0 \wedge y \ mod \ 2 = 0) \implies -Ax + B\frac{y}{2} > 0$ for all x. This implies that A must be 0 but B is a positive number, thus $B > 0$.

This information can be used to analyze the second branch: $(y > 0 \wedge y \ mod \ 2 \neq 0) \implies B\frac{y-1}{2} - Cx > 0$; this implies C to be 0. The template for the ranking function simplifies to $By, B > 0$ or simply y after normalizing it.

This approach works very well for termination analysis using polynomial ranking functions (which includes both linear and nonlinear), evaluating to natural numbers or integers bounded from below.

Consider however the following example which is quite interesting from the perspective of termination analysis.

Example 6. **var** n: integer **end var**
while $n > 1$ **do**
if $n \ mod \ 2 = 0$ **then** $n := \frac{n}{2}$ **else** $n := n + 1$;
end while

It can be shown that no polynomial ranking function can be used to show termination of the loop above even though the program does terminate as should be obvious to the reader. No ranking function involving exponentiation works either. However the function $n + 2(n + 1)$ *mod* 2 serves as a ranking function; such a template can be constructed/hypothesized by analyzing the program and finding all functions other than the polynomial operations appearing in it (such as *mod* 2) for possible use to generate templates.

6 Interpolant Generation using Quantifier Elimination

In [31], we were among the first ones to establish a direct connection between interpolant generation and quantifier elimination; in hindsight, this relationship seems obvious and trivial. To give a brief overview, given two formulas α and β such that $\alpha \implies \beta$, a Craig interpolant is a formula γ on symbols common to α as well as β such that $\alpha \implies \gamma \wedge \gamma \implies \beta$. Existence of such interpolants was proved by Craig [13]. If uncommon symbols can be eliminated from α (or equivalently β), an interpolant can be generated from the result. It is not essential for the underlying theory to admit full quantifier elimination for the quantifier-free theory of equality of uninterpreted function symbols: for example, there is no formula equivalent to $f(a) = b$ in which f does not appear.

In [31], we used quantifier elimination based approach for generating interpolants for quantifier-free theories of complex data structures such as finite lists, finite sets, finite arrays, finite bags, theory of free constructors, etc. Interpolants are generated from a combination of theories using Nelson and Oppen framework assuming each of the component theories have an algorithm for generating interpolants. In fact, Nelson and Oppen had insight that for convex theories it suffices to use equality interpolants on common variables [39].

We have been pursuing this approach based on quantifier elimination for developing algorithms for generating interpolants of varying strengths. It is easy to see that interpolants are closed under conjunction and disjunction: if γ_1 and γ_2 are interpolants of an (α, β) pair, then $\gamma_1 \wedge \gamma_2$ as well as $\gamma_1 \vee \gamma_2$ are also interpolants of (α, β). Interpolants for an (α, β) form a lattice under strict implication ordering with the strongest interpolant (which can be generated from α) and the weakest interpolant (which can be generated from β).

Many interpolant based methods for system analysis, for instance, methods for generating invariants, abstraction refinement in software model checking, generalization of Bradley's IC3 method for proving safety properties, depend upon the quality of interpolants generated. What interpolant is used can often determine the quality of invariant generated as well as affect the convergence of invariant generation algorithms. Very little is understood about the relationship between the kind of interpolants used and the performance and output of invariant generation algorithms.

As will be demonstrated below, in our approach, a proof of $\alpha \implies \beta$ is not needed whereas almost all methods proposed in the literature rely on a proof (either direct or refutation); further interpolants generated using these methods

not only differ in how they analyze proofs [15] but they also depends upon the proof being used (as there can be many proofs of $\alpha \implies \beta$).

As from α, interpolants can be generated from β by eliminating its uncommon symbols; there is an advantage in generating interpolants from α since all other interpolants can be computed by the implicating relation on α, which is easier than the inverse of implication relation. If the interpolant generated from α is the strongest, it can serve as an interpolant not only for β but the set of all formulas which are implied by α insofar as the only common symbols of such a formula with α remain the same. In other words, a single quantifier elimination on α can result in an interpolant that works for a family of many distinct β's. In that sense, even if quantifier elimination is expensive, its cost can often be amortized over a large class of such formulas implied by α. That can especially be useful if in an CEGAR like approach, refinements of abstractions only involve common symbols of α and β. Even if an incomplete quantifier elimination algorithm is used to perform quantifier elimination to generate an interpolant, it serves as an interpolant for all the formulas implied by the interpolant.

We illustrate some of these ideas for two quantifier-free subtheories: *theory of equality over uninterpreted symbols* and *theory of octagonal formulas over the integers, rationals or reals.*

6.1 Theory of equality of uninterpreted symbols (EUF)

This theory is interesting not only because it is extensively used in our method for generating interpolants for quantifier-free theories of container data structures [31], but also because an interpolation algorithm must eliminate uncommon function symbols.

The input to the algorithm is a satisfiable α which is assumed to be a conjunction of equations and disequations on ground terms involving constants and nonconstant function symbols (henceforth called function symbols contrasting with constants). There are three phases in the algorithm. The first phase is to generate congruence closure, much like in [33]. All uncommon constants which are equivalent to common constants are eliminated by substitution. At the end of this phase, equations and disequations can be divided into two parts: (i) those containing only common symbols, and (ii) those containing at least one uncommon symbol that cannot be eliminated just by substitution.

The second phase is the most interesting in which uncommon function symbols as well as uncommon constants appearing as arguments to even common function symbols are eliminated. The result of this phase is in general a collection of Horn clauses in which an uncommon symbol can be eliminated only under some conditions. This phase differs completely from a congruence closure algorithm.

The final phase is interpolant generation: two possibilities are discussed. In the first possibility, uncommon symbols are eliminated but in general, this step can lead to an exponential blow-up, both in the number of steps as well as the size of an interpolant generated. In the second possibility, uncommon symbols

are presented in a solved form. This step can be executed in polynomial time, much as phase 2 and phase 1.

1. **Flattening:** As in Kapur's congruence closure algorithm [33], flatten nonconstant terms by introducing new constants so that all equations and disequations in α are of the form:

$$f(c_1, \ldots c_k) = d, \; c = d \qquad \text{or} \qquad c \neq d.$$

   The first type of an equation is called an f-equation or a function equation; the second type of an equation is a constant equation.
   All the new constant symbols introduced to stand for flattened subterms (which are nodes of a dag representation of terms in α) are classified to be either common or uncommon based on symbols appearing in them. With innermost traversal of nonconstant subterms in α, any new constant symbol introduced to stand for a nonconstant term that includes an uncommon symbol is labeled uncommon. A new constant symbol introduced to stand for a nonconstant term consisting of common symbols only is labeled a common symbol. So a new constant is uncommon iff it stands for a flat term which has an uncommon symbol.
   If there is an f-equation $f(c_1, \cdots c_k) = d$ in which $f, c_1, \cdots c_k$ are common symbols and d is an uncommon symbol, then introduce a new common symbol, say e, and replace the above equation by two equations f-equation $f(c_1, \cdots c_k) = e$ purely in common symbols and $d = e$.
   It is easy to see that every f-equation has only common symbols or at least one of $f, c_1, \cdots c_k$ in the nonconstant term is an uncommon symbol.
2. **Phase I: Elimination of uncommon constants:** Process constant equations to generate equivalence classes of constants using union-find (or balanced tree data structure) to generate constant congruence. Pick from every equivalence class, a representative as follows: (i) if an equivalence class contains common constants, pick any common constant; (ii) if an equivalence contains only uncommon constants, then pick any uncommon constant. It is important to pick a common constant as a representative whenever that is possible.
   Replace each constant symbol in the disequations and remaining equations including f-equations by the representative of its equivalence class.
   This eliminates all uncommon constants which have common constants as their representatives. From an equivalence class which has an uncommon as well as a common constant, discard all uncommon constants as they are not needed in interpolant generation.
3. **Deduction of additional equalities:** If there are two f-equations with identical left sides, then add the equation generated by equating their right sides.
   Repeat Step 2 followed by Step 3 until no new equalities are generated.
   The result of these steps is the ground congruence closure including singleton equivalence classes.

4. **Phase II: Elimination of uncommon function symbols:** If a function symbol f is uncommon, then from every distinct pair of f-equations, generate a Horn clause as follows: Given two distinct equations $f(c_1, \ldots, c_k) = d$ and $f(e_1, \ldots, e_k) = h$, generate a Horn clause of the form
$(c_1 = e_1 \wedge \cdots \wedge c_k = e_k) \implies d = h$,
after normalizing, i.e., delete trivial equations in the hypothesis as well as delete the clause if the conclusion is trivial or in the equivalence relation generated by the hypotheses.
If $k = 0$, then update the equivalence relation on constant symbols repeatedly applying Steps 2 and 3 above. If $k > 0$ and the Horn clause has only common constants, then it is included in the interpolant, like other equations and Horn clauses with common symbols.
If a Horn clause has only common constants in the hypothesis but the conclusion only has a nonconstant symbol on one side, then it can be used as a rewrite rule to eliminate this uncommon symbol.
5. **Exposing uncommon constants underneath common function symbols:** Uncommon constants appearing as arguments in an f-equation can be eliminated by exposing even when f is a common symbol. For any pair of f-equations which have as arguments uncommon constants, generate a Horn clause as was done for uncommon function symbols above. In principle, this step can be performed irrespective whether any uncommon constant is an argument in an f-equation or not but we apply it only if there is an uncommon constant hiding under f as it can unnecessarily generate more complex interpolants, particularly Horn clause interpolants, which are not the strongest.
At this step, the result is: a subset of equations and disequations purely in common symbols, and hence a part of an interpolant, and a subset of equations, disequations and Horn clauses in which at least one uncommon symbol appears.
6. **Phase III: Eliminating Uncommon symbols conditionally**
At this step, uncommon symbols can only be eliminated conditionally, which can significantly contribute to the complexity of computing an interpolant. An alternative explored later is to keep such uncommon symbols in solved form so that they can be eliminated as required by an application in which interpolants are being used.
7. **Deleting uncommon constants that cannot be eliminated**
If for some uncommon constant, there is no Horn clause with a conclusion relating this uncommon symbol to some other symbol, then eliminate all clauses in which this uncommon symbol appears since these clauses cannot be used any further to generate an interpolant.
8. **Eliminating uncommon constants by conditional rewriting:** Using a conditional equation $(c_1 = e_1 \wedge \cdots \wedge c_k = e_k) \implies d = h$, where d (equivalently, h) is the only uncommon symbol and all other symbols are common, d (respectively, h) can be conditionally eliminated from other equations, conditional equations and disequations in which d appears. For every such conditional equation which can eliminate d, such rewriting must be performed. It is easy to see that this can lead to an exponential blow-up. If there is only

one such conditional equation for d, then the exponential blow-up can be avoided as such a case is not much different from unconditional elimination. As in the case of unconditional elimination, replacing an uncommon symbol can lead to new equalities as well as conditional equations. The above two steps are then repeated until no uncommon symbol can be completely eliminated or there are no Horn clauses left relating uncommon symbols.

9. **Generating an interpolant:** The result of the above steps after repeated applications is: (i) the set of equations, disequations and conditional equations purely in common symbols, (ii) conditional equations each of which has an uncommon symbol in its hypothesis which has been eliminated elsewhere. An interpolant is the set of equations, Horn clauses and disequations purely in common symbols including new common symbols. This representation generates compact interpolants.

An interesting example The above algorithm is illustrated using the following example from [20] in which $\alpha = \{f(z_1, v) = s_1, f(z_2, v) = s_2, f(f(y_1, v), f(y_2, v)) = t\}$ in which v is the only uncommon symbol and f as well as constants $z_1, z_2, y_1, y_2, s_1, s_2, t$ are common.

After flattening, $\alpha = \{f(z_1, v) = s_1, f(z_2, v) = s_2, f(y_1, v) = n_1, f(y_2, v) = n_2, f(n_1, n_2) = t\}$ with n_1, n_2 being the new uncommon symbols along with v from the input.

Since there are no constant equations, no equivalence classes on constants are generated (which is the same as each equivalence class containing a singleton constant).

Even though f is a common function symbol, uncommon symbols are hiding under f as its arguments in f-equations. Applying the step to expose uncommon symbols, we have, Horn clauses generated from the above rules:

$\{1.\ z_1 = z_2 \implies s_1 = s_2,\ 2.\ z_1 = y_1 \implies n_1 = s_1,\ 3.\ z_1 = y_2 \implies n_2 = s_1,\ 4.\ (z_1 = n_1 \wedge v = n_2) \implies s_1 = t,\ 5.\ z_2 = y_1 \implies n_1 = s_2,\ 6.\ z_2 = y_2 \implies n_2 = s_2,\ 7.\ (z_2 = n_1 \wedge v = n_2) \implies s_2 = t,\ 8.\ y_1 = y_2 \implies n_2 = n_1,\ 9.\ (y_1 = n_1 \wedge v = n_2) \implies n_1 = t,\ 10.\ (y_2 = n_1 \wedge v = n_2) \implies n_2 = t\}$.

An analysis of the above Horn clauses reveals that while there are Horn clauses to rewrite and eliminate uncommon symbols n_1, n_2 but there is none for eliminating v. Because of this observation, all equations and Horn clauses in which v appear can be deleted, and the only 4th equation in the input along with Horn clauses $1, 2, 3, 5, 6, 8$ are left.

Both n_1 and n_2 can be successively eliminated. The second Horn clause has only common symbols in the hypothesis and the conclusion has an uncommon symbol that can be replaced by a common symbol. So this conditional rewrite can be applied wherever n_1 appears. Below, only some relevant steps are shown.

2. $z_1 = y_1 \implies n_1 = s_1$ simplifies n_1 to s_1 under $z_1 = y_1$. Horn clause 5 simplifies giving 11. $(z_1 = y_1 \wedge z_2 = y_1) \implies s_1 = s_2$; This Horn clause is subsumed by 1 and hence is deleted.

8 simplifies to 12. $(y_1 = y_2 \wedge z_2 = y_1) \implies n_2 = s_1$; the equation $f(n_1, n_2) = t$ simplifies to 13. $z_1 = y_1 \implies f(s_1, n_2) = t$.

Clause 3 is used to simplify: Clause 6 simplifies to $(z_2 = y_2 \wedge z_1 = y_2) \implies s_1 = s_2$ which is subsumed by 1. 12 simplifies to: 14. $(y_1 = y_2 \wedge z_1 = y_2 \wedge z_2 = y_1) \implies s_1 = s_1)$ which is discarded. The f-equation 13 simplifies to $(z_2 = y_1 \wedge z_1 = y_1) \implies f(s_1, s_2) = t$. And so on.

After replacement of n_1, n_2 using Horn clauses 2, 3, 5, and 6, the result purely in common symbols is:

$\{1.\ z_1 = z_2 \implies s_1 = s_2, (z_1 = y_1 \wedge z_2 = y_2) \implies f(s_1, s_2) = t, (z_2 = y_1 \wedge z_1 = y_2) \implies f(s_2, s_1) = t, (z_1 = y_1 \wedge z_1 = y_2) \implies f(s_1, s_1) = t, (z_2 = y_1 \wedge z_2 = y_2) \implies f(s_2, s_2) = t\}$, which is the interpolant generated from α by removing an uncommon symbol v.

The above example can be generalized to: $\alpha = \{f(z_1, v) = s_1, f(z_2, v) = s_2, \cdots, f(z_k, v) = s_k,\ g(f(y_1, v), f(y_2, v), \cdots, f(y_k, v)) = t\}$ with the only uncommon symbol v and $z_1, \cdots, z_k, y_1, \cdots, y_k, s_1, s_2, \cdots s_k, t$ are constant common symbols. α is of size $O(k)$. A pure interpolant would be of exponential size and is of the form $\bigwedge(z_{j_1} = y_{i_1} \wedge z_{j_2} = y_{i_2} \cdots \wedge \cdots z_{j_k} = y_{i_k}) \implies g(s_{u_1}, \cdots, s_{u_k}) = t$ for various possible sets of $s_1, \cdots, s_k$.

It may however be possible to avoid exponential blow-up by encoding non unary function symbols by currying them. This aspect would be investigated in a forthcoming paper [30].

Pseudo-Interpolants In order to avoid exponential blow-up. we introduce *pseudo-interpolants* which can have uncommon symbols with conditional substitutions for them in a solved form, very similar to solved form for substitutions in unification problems. The situation here is more complex because of conditional substitutions whereas in the case of standard unification (over the empty theory), complete subterm sharing using a dag representation of fully shared subterms suffices to avoid exponential blow-up.

A pseudo-interpolant is a finite set of equations and conditional equations purely in common symbols along with a finite set of conditional substitutions and a total (could be partial) ordering on uncommon symbols based on the conditional dependency of uncommon symbols among themselves. The idea here is to determine the order in which uncommon symbols would be conditionally eliminated by Horn clauses in the conditional elimination step above. More details will be provided in a forthcoming paper [30].

Pseudo interpolants for the example discussed in the previous subsection would be the rules:$\{1, 2, 3, 5, 6, 8\}$ and $\{f(n_1, n_2) = t\}$ with the ordering $n_2 > n_1$ in which $2, 5$ are used to eliminate n_1 followed by $3, 6$ to eliminate n_2; the ordering $n_1 > n_2$ also works. The result would be the same as the interpolant given in the previous subsection.

v A pseudo-interpolant is thus an intermediate form where some uncommon symbols are eliminated only if needed in an application where interpolants are needed.

6.2 Complexity Analysis

The flattening step can be done in $O(n)$ where n is the size of the graph representation of the input terms (with full sharing). In general there are $O(n)$ constant symbols after flattening, corresponding to a constant for every node in the graph representation.

The constant congruence step and associated processing of replacing constants by their representatives can be done easily in $O(n * log(n))$ (but its amortized complexity is $O(n * \alpha(n))$, almost linear, since α here is the inverse of Ackermann's function.

Horn clauses of size k, where k is the maximum arity of a function symbol, (or constant size if all nonunary function symbols are encoded using the single binary function *apply* as proposed in [40]) can be generated in $O(n^2)$ steps.

The most expensive step is that of conditional rewriting primarily because for a single uncommon symbol, there can be multiple Horn clauses equating the uncommon symbol under different conditions. This can result in an exponential blow-up if uncommon symbols are completely eliminated; the above example illustrates this: for n_1 there are two conditional Horn clauses used to eliminate it; similarly n_2 is eliminated by two conditional Horn clauses.

In contrast, a pseudo-interpolant can be generated in $O(n^3)$. Perhaps this complexity can be improved substantially by exploiting the structure of Horn clauses. But a (pure) interpolant only in common symbols can require exponentially many steps as the following example illustrates.

To our knowledge, this is the first complexity analysis of interpolant generation algorithms in the literature. Furthermore, the above complexity results are for generating the strongest possible interpolants from α without having access to a proof of $\alpha \implies \beta$ (which can in general be of higher complexity than interpolant generation if the complexity of proof generation is also included in the complexity analysis of interpolant generation).

Another interesting byproduct of the proposed approach is that a conditional congruence (completion) relation algorithm can be generated from a finite set of conditional Horn clauses from which another Horn clause can be easily decided by simplification. A paper explaining the algorithm is under preparation.

Some examples from the literature on Interpolant generation for EUF
There are many algorithms proposed in the literature for generating interpolants from refutation proofs of $\alpha \wedge \neg\beta$ including those in [19, 37, 17].

To compare the interpolants generated from the above algorithm with those in the literature (particularly McMillan's [37] and Tinelli et. al's [17]), consider Example 3.1 in Tinelli et al's paper. $\alpha = \{z_1 = x_1, x_1 = z_2, z_2 = x_2, x_2 = f(z_3), f(z_3) = x_3, x_3 = z_4, f(z_2) = x_2, x_2 = z_3\}$ and $\beta = \{z_1 = y_1, y_1 = f(z_2), f(z_2) = y_2, y_2 = z_3, z_3 = y_3, z_2 = y_2, y_2 = f(z_3), y_3 \neq z_4\}$.

Common symbols are $\{f, z_1, z_2, z_3, z_4\}$. So other symbols from α must be eliminated which are all nonfunction symbols, so it can be done easily by congruence closure on constants.

The constant congruence gives equivalence classes: $\{z_1, x_1, z_2, x_2, z_3\}$ and $\{x_3, z_4\}$; wlog, let z_1 and z_4 respectively be the representative. In f-equations, constants are replaced by their representatives: $\{f(z_1) = z_1, f(z_1) = z_4, f(z_1) = z_1\}$, from which the equality $z_1 = z_4$ is deduced merging the two equivalence classes. x_1, x_2, x_3 are then deleted from the equivalence classes.

Deleting uncommon symbols, the interpolant is: $\{z_2 = z_1, f(z_1) = z_1, z_4 = z_1, z_3 = z_1\}$, whereas McMillan's algorithm (as reported in Tinelli et al's paper) produces $\{z_1 = z_2, z_2 = f(z_3), f(z_3) = z_4\}$, and Tinelli et al's algorithm gives $\{z_1 = z_4\}$. The interpolant generated by our algorithm implies the interpolants generated by both McMillan's as well as Tinelli et al's algorithms and is stronger.

Also consider Example 4.3 in Tinelli et al's paper: $\alpha = \{x_1 = z_1, z_2 = x_2, z_3 = f(x_1), f(x_2) = z_4, x_3 = z_5, z_6 = x_4, z_7 = f(x_3), f(x_4) = z_8\}$ and $\beta = \{z_1 = z_2, z_5 = f(z_3), f(z_4) = z_6, y_1 = z_7, z_8 = y_2, y_1 \neq y_2\}$.

Commons symbols are $\{f, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$ and $\{x_1, x_2, x_3, x_4\}$ are uncommon symbols to be eliminated from α.

Using our algorithm, the constant equivalence relation is:
$\{\{x_1, z_1\}, \{x_2, z_2\}\{x_3, z_5\}, \{x_4, z_6\}\{z_3\}\{z_4\}, \{z_7\}, \{z_8\}\}$, let $z_1, z_2, z_5, z_6, z_3, z_4, z_7, z_8$ respectively be the representatives of the equivalence classes. Replacing constants in f-equations gives: $f(z_5) = z_7, f(z_6) = z_8, f(z_1) = z_3, f(z_2) = z_4$,.

There is no need to generate any Horn clauses since none of the f-rules has any uncommon symbol. The interpolant is: $\{f(z_1) = z_3, f(z_2) = z_4, f(z_5) = z_7, f(z_6) = z_8\}$. The interpolant reported for McMillan is: $(z_1 = z_2 \wedge (z_3 = z_4 \implies z_5 = z_6)) \implies (z_3 = z_4 \wedge z_7 = z_8)$, whereas for Tinelli et al's algorithm, it is: $(z_1 = z_2 \implies z_3 = z_4) \wedge (z_5 = z_6 \implies z_7 = z_8)$.

Our interpolant uses f since it is a common symbol as well. Suppose we wanted an interpolant without f, then we will generates Horn clauses thus eliminating f to give: $(z_1 = z_2 \implies z_3 = z_4) \wedge (z_1 = z_5 \implies z_3 = z_7) \wedge (z_1 = z_6 \implies z_3 = z_8) \wedge (z_2 = z_5 \implies z_4 = z_7) \wedge (z_2 = z_6 \implies z_4 = z_8) \wedge (z_5 = z_6 \implies z_7 = z_8)$. However, this unnecessarily increases the size of the interpolant as well as generating a weaker interpolant whereas this rule is necessary if the function symbol is uncommon and must be eliminated.

Example in Figure 4 of Tinelli et al's paper [17]: $\alpha = \{x_1 = z_1, z_3 = f(x_1), f(z_2) = x_2, x_2 = z_4\}$ and $\beta = \{z_1 = y_1, y_1 = z_2, y_2 = z_3, z_4 = y_3, f(y_2) \neq f(y_3)\}$. Common symbols are: $\{f, z_1, z_3, z_3, z_4\}$. Constant congruence is $\{\{x_1, z_1\}, \{x_2, z_4\}\}$. Replacing constants by representatives in f-equations gives $\{f(z_1) = z_3, f(z_2) = z_4\}$. The interpolant I_α is thus $\{f(z_1) = z_3, f(z_2) = z_4\}$, the same as the one produced by McMillan's algorithm; in contrast, Tinelli et al produced $z_1 = z_2 \implies z_3 = z_4$ which would be generated by our algorithm if an interpolant without f is desired.

As the above examples illustrate, our algorithm produces the strongest interpolant and is also theoretically more efficient.

6.3 Interpolant Generation for Octagonal Formulas

In this section, we use quantifier elimination to generate interpolants for a conjunction of octagonal formulas over integers (or reals or rationals).

Consider α, a finite satisfiable conjunctions of $a\ x_i + b\ y_i \leq c$, where $a, b \in \{-1, 0, -1\}$ but $c \in \mathbb{Z}$. Given a set of symbols x_i's and y_j's declared to be local to α, our goal is to eliminate them from α to get a projection which is a formula purely in the remaining symbols. This is done by eliminating one symbol at a time and generating a new set of octagons formulas as follows[12]:

Elim x_i: For every pair of octagon atoms $a\ x_i + b\ y_j \leq c$ and $-a\ x_i + b'\ y_k \leq d$, generate a new octagon literal by adding them, thus eliminating x_i: $b\ y_I + b'\ y_j \leq c + d$. This is done for all such pairs including symbol constraints in which y_k does not appear.

Normalization: In the special case when $y_i = y_j$, the above can give $2y_i \leq c+d$ or $-2y_i \leq c + d$ which must be normalized by dividing $c + d$ by 2 and taking the integer floor (in case of computing a projection over the reals or rationals, simple divison is performed). This inference is also called tightening.

Repeat this process until all local symbols are eliminated generating an interpolant I_α from α. Before applying the above rules, preprocessing is performed to simplify $x - x \leq c$ to T if c is nonnegative, and F otherwise; $b\ x - b'\ y \leq c \wedge b\ x - b'\ y \leq d$ is simplified to $b\ x - b'\ y \leq min(c, d)$. After the uncommon symbols are eliminated and octagonal formulas in which uncommon symbols appear are removed, the result is an interpolant.

Let α have n symbols and m octagonal atoms, then it puts an upper bound on the number of octagonal atoms which are $O(n^2)$. If k symbols need to be eliminated, the worst case complexity of interpolant generation is $O(k * n^2)$. Many heuristics are possible which can reduce the complexity further.

We discuss below the case when octagonal formulas are over integers as that is more interesting.

Comparison with Griggio's Algorithm Below, we illustrate the algorithm on examples mostly taken from Griggio's thesis [19] (see also [5]) and show differences between their algorithm and our algorithm. Griggio performed a detailed analysis of an unsatisfiable set of octagonal formulas and considered its different partitions into α and β to illustrate the intricacies of his graph based algorithm. Following Miné who introduced two variables x^+ and x^- for every variables x to stand for $+x$ and $-x$, to respectively and transforming every octagonal formula into two difference constraints of the form $u - v \leq c$, a set of octagonal formulas can be represented as a weighted difference graph. If this graph has a negative cycle, then the constraint set is unsatisfiable. Even when the graph has a 0 weight cycle, then sometimes formulas can be unsatisfiable over the integers.

Without giving all the details of Griggio et al's algorithm, some characteristics of Griggio's examples are (i) when represented as a graph of difference constraints, it is unsatisfiable but with a 0 weight cycle and (ii) his algorithm generates a conditional interpolant for some partitions even though the same

[12] It will be interesting to develop an efficient method for simultaneously eliminating multiple symbols; as of now we have not succeeded in developing such a method.

refutation proof is used. Griggio showed the behavior of his algorithm to generate interpolants based on different subsets of common symbols of various partitions.

Consider Ex. 4.26 of the two formulas from Griggio's thesis [19] (pp. 146-147) (Example 5 in [5]): $\alpha = \{x_1 - x_2 \geq -4,\ \ -x_2 - x_3 \geq 5,\ \ x_2 + x_6 \geq 4,\ \ x_2 + x_5 \geq -3\}$ and $\beta = \{\ -x_1 + x_3 \geq -2,\ \ -x_4 - x_6 \geq 0,\ \ -x_5 + x_4 \geq 0\}$.

In contrast, in our approach, we identify x_2 as the symbol to be eliminated from α, since it is local to α only. We do not care of a refutation proof of $\alpha \wedge \beta$. Using the *Elim* rule to eliminate x_2 from complimentary literals, we get $\{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7,\ \ -x_3 + x_6 \geq 9\}$, leaving no other pairs of octagon formulas with a positive x_2 and a negative x_2. Since every literals of α includes x_2, nothing from α is included in the interpolant, with the result being $I_\alpha = \{-x_3 + x_5 \geq 2, x_1 + x_6 \geq 0, x_1 + x_5 \geq -7, -x_3 + x_6 \geq 9\}$. It can be checked that I_α is implied by α and is inconsistent with β.

Griggio's algorithm generated a conditional interpolant: $(-x_6 - x_5 \geq 0) \implies (x_1 - x_3 \geq 3)$, which is implied by I_α since $-x_6 - x_5 \geq 0$ with $x_1 + x_5 \geq -7 \wedge x_1 + x_6 \geq 0$ with tightening gives $x_1 \geq -3$; similarly, $-x_6 - x_5 \geq 0$ with $-x_3 + x_5 \geq 2 \wedge -x_3 + x_6 \geq 9$ with tightening gives $-x_3 \geq 6$; from $x_1 \geq -3 \wedge -x_3 \geq 6$, we get $x_1 - x_3 \geq 3$. Further, the interpolant generated by the proposed algorithm is strictly stronger than Griggio's interpolant. Griggio's observation on p. 146 is correct that the first two octagonal formulas in our interpolant is not an interpolant since it is not inconsistent with β even though they are implied by α but including two additional octagonal formulas shown above produces the strongest interpolant. We consider this as a major weakness of proof-based interpolant generation algorithms.

Other Examples In this section, we discuss almost all the examples from [19, 5] to illustrate differences as well as superiority of our algorithm particularly the quality of interpolants generated.

Consider example 1 in [5] (also Ex. 4.17 in [19]) which can be done over the rationals because of a negative weight cycle, and thus does not require any complex analysis of the negative cycle.

$\alpha = \{-x_2 - x_1 + 3 \geq 0, x_1 + x_3 + 1 \geq 0, -x_3 - x_4 - 6 \geq 0, x_5 + x_4 + 1 \geq 0\}$ and $\beta = \{x_2 + x_3 + 3 \geq 0, x_6 - x_5 - 1 \geq 0, x_4 - x_6 + 4 \geq 0\}$. The uncommon symbol to be eliminated from α is x_1: its elimination gives an interpolant: $I_\alpha = \{-x_2 + x_3 + 4 \geq 0, -x_3 - x_4 - 6 \geq 0, x_5 + x_4 + 1 \geq 0\}$.

If we eliminate the uncommon symbol x_6 from β: $I_\beta = \neg(x_2 + x_3 + 3 \geq 0 \wedge x_4 - x_5 + 3 \geq 0)$. In contrast, Cimatti et al. reported $(-x_2 - x_4 - 2 \geq 0 \wedge x_5 - x_3 - 5 \geq 0)$, which is strictly implied by I_α and hence weaker than I_α, and which also implies I_β. Griggio reported in his thesis (p. 135) that McMillan's algorithm for linear arithmetic over the rationals would generate an even more complicated interpolant which is not even an octagon: $-x_2 - x_4 + x_5 - x_3 - 7 \geq 0$ which is also strictly implied by the interpolant generated by our algorithm. Further, in contrast to McMillan's algorithm as well as Griggio's algorithm, our algorithm will always generate a conjunction of octagoal formulas as an interpolant. This demonstrates that it is often possible to devise more efficient

quantifier elimination heuristics for subtheories producing succinct and elegant results.

Let us now consider the examples used by Griggio et. al. [5] to illustrate different case analysis of 0-weight cycles. The case above in our view, illustrates the most complex case. The other three cases, which are relatively easy, are illustrated below.

Example 2 (Example 4.20 in [19]): $\alpha = \{x_3 - x_1 \geq -2, -x_6 - x_4 \geq 0, x_4 - x_5 \geq 0\}$, and $\beta = \{x_1 - x_2 \geq -4, \quad -x_2 - x_3 \geq 5, \quad x_2 + x_6 \geq 4, \quad x_2 + x_5 \geq -3\}$, reversing α and β from the running example. x_4 is eliminated from α to give: $x_3 - x_1 \geq -2, -x_5 - x_6 \geq 0$ as I_α; this result is the same as in [19].

Example 3 (Example 4.22 in [19]) : $\alpha = \{x_1 - x_2 \geq -4, \quad x_2 + x_6 \geq 4, \quad -x_4 - x_6 \geq 0, \quad -x_1 + x_3 \geq -2\}$ and $\beta = \{ \quad -x_2 - x_3 \geq 5, \quad x_2 + x_5 \geq -3, \quad x_4 - x_5 \geq 0\}$. The symbols local to α are x_1, x_6. Eliminating them gives: $I_\alpha = \{x_3 - x_2 \geq -6, x_2 - x_4 \geq 4\}$, again the same as reported by [5].

Example 4: $\alpha = \{x_1 - x_2 \geq -4, \ x_2 + x_6 \geq 4, \ -x_1 + x_3 \geq -2, \ -x_2 - x_3 \geq 5\}$ and $\beta = \{ \ x_2 + x_5 \geq -3, \quad -x_5 + x_4 \geq 0, \quad -x_4 - x_6 \geq 0\}$. The symbols local to α are x_1, x_3. Eliminating them gives: $I_\alpha = \{-x_2 \geq 0, x_2 + x_6 \geq 4\}$; the result reported in [5] is $-x_2 + x_6 \geq 4$, which is a weaker interpolant.

The following example 4.24 from [19] can be quite revealing as well: $\alpha = \{x_1 - x_2 \geq -4, \ x_2 + x_6 \geq 4, \ -x_1 + x_3 \geq -2, \ -x_2 - x_3 \geq 5, \ x_2 + x_5 \geq -3\}$ and $\beta = \{-x_6 - x_4 \geq 0, \ x_4 - x_5 \geq 0\}$ with common symbols being x_5, x_6. Eliminating x_1, x_2, x_3 one by one, gives $\{x_6 \geq 4, x_5 \geq -3\}$, which is the strongest interpolant generated from α. In this case, since β is smaller and simpler and furthermore, only one symbol x_4 needs to be eliminated, so an interpolant $\neg(-x_6 - x_5 \geq 0)$, which is equivalent to $x_5 + x_6 > 0$, can be easily generated which is weaker than the one generated from α. Clearly, the interpolant from α which are conditions on single variables is more likely to be useful and efficient in applications.

In our approach, an interpolant generated is always a conjunction of octagonal formulas. It is easy to see that (i) the strongest interpolant is also a conjunction of octagonal formulas and further, (ii) our algorithm generates this octagonal formula as an interpolant. More details will be provided in a forthcoming paper [30].

This algorithm is typically faster since it is only considering α and not attempting to generate a refutational proof of $\alpha \wedge \beta$ and more importantly, generates a simpler interpolant for a family of β's. As stated above, the worst case complexity of the algorithm is $O(k * n^2)$ where k symbols need to be eliminated from n symbols in α. Typically the number of octagonal atoms in α is much smaller than $O(n)$ in which the complexity is $O(k * m^2)$.

6.4 Interpolant generation for concave quadratic nonlinear polynomial constraints using linearization

In [18] we have developed an efficient method for generating quadratic polynomial interpolants for concave quadratic polynomial formulas (both pure as well as combined with uninterpreted symbols). The key idea is to generalize Mozkin's transposition theorem using a linearization technique. This is an illustration of

using specialized heuristics for quantifier elimination for a subclass of nonlinear polynomial constraints under certain conditions. An efficient version has also been developed and implemented using semi-definite programming thus combining symbolic and numerical techniques. More details can be found in [18] for a theoretical framework and for a preliminary implementation which can handle almost all numeric relational abstract domains found useful in the abstract interpretation approach. More details and its possible applications can be found in [18, 50, 49]

7 Generating Abductors in Saturation based approach for strengthening Invariants

In [16], we have developed a saturation based approach for checking whether a given annotation at a program location indeed holds. The technique attempts to use the annotation in verification conditions of the program to establish that they are preserved over all possible program paths (or can lead to proofs of other annotations in the program). Consider the case of an annotation inside a loop body with a loop invariant as being an instance. To check whether such an annotation is indeed invariant, one way is to use the inductive assertion approach advocated by Floyd and Hoare, and check whether all basic cycles through that annotated location inside the loop body preserve the assertion.

Consider the following simple example that is quite instructive.

Example 7. **var** x, y, z: integer **end var**
$x := 0, \quad y := 0, \quad z := 9;$
while $x \leq N$ **do**
$x := x + 1; \quad y := y + 1; \quad z := z + x - y;$
end while

Then $z \leq 0$ is a loop invariant. Since $z \leq 0 \implies z + x - y \leq 0$, it is not inductive. It can, however, be strengthened to $z \leq 0 \wedge x - y \leq 0$ which is true initially as well as preserved by the body of the loop.

The saturation based approach for checking whether a given annotation α is a loop invariant, it attempts to show that the annotation α is inductive by generating verification condition of the form $(\alpha \wedge cond) \implies \beta$, every possible possible basic cycle through the loop entry, where *cond* is either true or a formula generated from tests along that cycle. Since that is not so in this case, it attempts to find ψ, which we call *abductor* of $(\alpha \wedge cond)$ and β , such that $((\alpha \wedge cond) \wedge \psi) \implies \beta$; there can in general be infinitely many such ψ's with the simplest trivial one being $false$ which gives little information about the behavior of the loop. However, there are other nontrivial abductors including $x - y \leq 0$ using which $z \leq 0$ can be shown to an invariant since $z \leq 0 \wedge x - y \leq 0$ is inductive.

Quantifier elimination can be used to find such abductors as follows: To show $(\alpha \wedge \psi) \implies \beta$ is valid (after abstracting *cond* to be true as above) equivalent

to computing ψ that implies $\alpha \implies \beta$. Eliminating some common variables from $\alpha \implies \beta$ can give a simpler formula which ψ must imply. For the above example, ψ must imply $(z \leq 0) \implies (z + x - y \leq 0)$; eliminating z from $\neg(z \leq 0 \wedge (y - x - z) < 0$ gives $\neg(y - x < 0)$ which is $y - x \geq 0$, equivalently $x - y \leq 0$; any ψ that is stronger than $x - y \leq 0$ is such an abductor including $x - y \leq 0$ itself.

We are actively exploring this approach using such approximations based on quantifier elimination heuristics.

8 Challenges for Symbolic Computation and SMT Communities

The first part of this paper reviewed our research on generating linear and nonlinear polynomial relations as invariants of programs; both approaches discussed above invariants rely on symbolic computation algorithms of high complexity – typically doubly exponential in the number of variables (and degree of polynomials). While many ideal theoretic operations needed in the ideal-theoretic approach can be implemented using Gröbner basis algorithm, there is never any need to compute a complete Gröbner basis for invariant generation given that any basis of the ideal of invariants suffices. Wheres Gröbner basis computations resulting from invariant generation for many nontrivial examples can be easily calculated as demonstrated in [44], complete quantifier elimination algorithms for the theory of real closed fields based on cylinderical algebraic decompositions (CAD) as implemented in QEPLOG or REDLOG do not work at all even for very simple examples such as the ones discussed in earlier sections. We have experienced similar problems in our attempts to use QEPLOG and REDLOG for generating invariants for hybrid system problems (such as the oil-pump problem) or for interpolant generation for nonlinear polynomial constraints.

As demonstrated, many other program analysis problems can be formulated as quantifier elimination problems. This includes termination analysis by generating ranking functions, invariant generation using counter example guided abstraction refinement approach [6, 1], a generalization of IC^3 approach considered extremely effective for hardware verification [3, 21, 2, 4] where interpolants need to be generated, abductive inference methods for invariant generation as in [16] for inductive invariant generation.

8.1 Addressing the Complexity Barrier

As we have tried to suggest above, many problems related to static analysis of loop programs can be formulated in terms of quantifier elimination. However, quantifier elimination is derided by our community because of high complexity of elimination methods be they on an algebraically closed field or the real closed field. It is our contention that most researchers nevertheless end up proposing heuristics for restricted quantifier elimination disguising them under different

often fancy terminology. To put it more mildly, we have yet to see any new proposal that is totally independent of the use of quantifier elimination. In this note, we have **called a spade a spade** by opening admitting that our approaches heavily rely on quantifier elimination; however, we are always attempting to find heuristics exploiting the special structure of formulas being addressed and living with incompleteness. In [34] as well as [32], we have shown how quantifier elimination problems can be effectively solved even by hand for many formulas arising in quantifier elimination based approach for automatically generating loop invariants. That is in sharp constrast to our unsuccessful attempts to use quantifier elimination software tools, especially for the theory of real closed field, even for the case when parametric shapes are linear constraints with parameters. For polynomial equalities, even parametric Gröbner basis algorithms appear to be an overkill.

As shown in [34, 32], it often suffices to find sufficient conditions on parameters which ensure that the verification conditions arising from all the paths are valid; such conditions, while incomplete, are often strong enough to generate meaningful invariants insofar as sufficient conditions are not too weak. For octagonal invariants, results using our approach are comparable or sometimes even better than those produced by the tool *Interproc* based on the abstract interpretation approach [23].

Approximating program behavior to ensure that verification conditions generated can be efficiently analyzed pays off; this is also the case in the abstract interpretation framework for generating transfer functions.

References

1. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.
2. Nikolaj Bjørner and Arie Gurfinkel. Property directed polyhedral abstraction. In *Proc. of VMCAI 2015*, pages 263–281, 2015.
3. Aaron R. Bradley. Understanding ic3. In *Proc. of SAT, 2012*, pages 1–14, 2012.
4. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.
5. Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Interpolant generation for UTVPI. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2009.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

7. M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Computer-Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.
8. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. *Programming Languages and Systems*, pages 140–140, 2005.
10. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.
12. Cox, D. and Little, J. and O'Shea, D. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, 1998.
13. William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.
14. David Cyrluk and Deepak Kapur. Reasoning about nonlinear inequality constraints: a multi-level approach. In *Proc. of Image Understanding Workshop*, pages 904–914, 1989.
15. Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, pages 129–145, 2010.
16. S. Falke and D. Kapur. When is a formula a loop invariant? In *Logic, Rewriting and Concurrency: Essays dedicated to Jose Meseguer on the Occasion of His* 65^{th} *Birthday*, LNCS 9200, pages 264–286. Springer-Verlag, 2015.
17. Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstic, and Cesare Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.
18. Ting Gan, Liyun Dai, Bican Xia, Naijun Zhan, Deepak Kapur, and Mingshuai Chen. Interpolant synthesis for quadratic polynomial inequalities and combination with euf. In *Proceedings, International Joint Conference on Auomated Reasoning (IJCAR) 2016*, Lecture Notes in Computer Science, pages 195–212. Springer, 2016.
19. Alberto Griggio. An effective smt engine for formal verification. *Ph.D. Thesis, University of Trento*, December, 2009.
20. Sumit Gulwani and Madan Musuvathi. Cover algorithms and their combination. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 193–207, 2008.
21. Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proc. of SAT, 2012*, pages 157–171, 2012.
22. J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Beyond finite domains. In *Principles and Practice of Constraint Programming*, pages 86–94. Springer, 1994.

23. B. Jeannet, M. Argoud, and G. Lalire. The interproc interprocedural analyzer.
24. D. Kapur. Geometry theorem proving using Hilbert's Nullstellensatz. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC 86)*, pages 202–208, 1986.
25. D. Kapur. An approach for solving systems of parametric polynomial equations. In Saraswat and Van Hentenryck, editors, *Principles and Practices of Constraint Programming*, pages 217–244. MIT Press, 1995.
26. D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. Technical report, Department of Computer Science, University of New Mexico, Albuquerque, NM, USA, 2003.
27. D. Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.
28. D. Kapur. Elimination techniques for program analysis. In *Proceedings of Harald Ganzinger Memorial Workshop, 2006*. Springer, 2012.
29. D. Kapur. Program analysis using quantifier elimination heuristics. In *Proc. of 12th Annual Conference on Theories and Models of Computation (TAMC)*, number 7287 in Lecture Notes in Computer Science, pages 94–109, 2012.
30. D. Kapur. Interpolant generation using quantifier elimination for euf and utvi formulas. Technical report, Dept. of Computer Science, University of New Mexico, Oct 2017.
31. D. Kapur, R. Majumdar, and C. Zarba. Interpolation for data structures. In *Proceedings of the 14th ACM SIGSOFT Symp. on Foundations of Software Engineering*, 2006.
32. D. Kapur, Z. Zhang, M. Horbach, H. Zhao, Q. Liu, and V. Nguyen. Goemetric quantifier elimination heuristics for automatically generating octagonal and max-plus invariants. In *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, number 7788 in Lecture Notes in Artificial Intelligence, 2010.
33. Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proc. Rewriting Techniques and Applications, 8th Intl. Conf., RTA-97*, pages 23–37, Sitges, Spain, June 1997. Springer LNCS 1231.
34. Deepak Kapur. A quantifier elimination based heuristic for automatically generating inductive assertions for programs. *J. of Systems Science and Complexity*, 19(3):307–330, 2006.
35. Deepak Kapur. A logic for parameterized octagonal constraints with $\pm\infty$. Oct. 2013.
36. Deepak Kapur and R. K. Shyamasundar. Synthesizing controllers for hybrid systems. In *Proceedings of Hybrid and Real-Time Systems, International Workshop. HART'97*, Lecture Notes in Computer Science, pages 361–375, Grenoble, France, March 1997. Springer.
37. Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
38. A. Miné. Weakly relational numerical abstract domains. *These de doctorat en informatique, École polytechnique, Palaiseau, France*, 2004.
39. G. Nelson. *Techniques for Program Verification*. PhD thesis, Department of Computer Science, Stanford University, Palo Alto, CA, 1981.
40. Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.
41. E. Rodríguez-Carbonell. *Automatic Generation of Polynomial Invariants for System Verification*. PhD thesis, Universitat Politecnica de Catalunya, 2006.

42. E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. *Intl. Symp. on Symbolic and Algebraic Computation (ISSAC)*, pages 266–273, July 2004.
43. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *J. of Science of Programming*, 64(1):54–75, 2007.
44. E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. of Symbolic Computation*, 42(4):443–476, 2007.
45. E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. *11th Symposium on Static Analysis (SAS)*, pages 280–295, August 2004.
46. S. Sankaranarayanan, H.B. Sipma, and Manna Z. Non-linear Loop Invariant Generation using Gröbner Bases. *Symp. on Principles of Programming Languages*, 2004.
47. H. Sheini and K. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability Testing*, pages 68–81. Springer, 2005.
48. W. Wu. Basic principles of mechanical theorem proving in geometries. *J. of Automated Reasoning*, 2:221–252, 1986.
49. H. Zhao, N. Zhan, and D. Kapur. Synthesizing switching controllers for hybrid systems by generating invariants. In *Proc. Festchrift Symp. in honor of He Jifeng*, pages 354–373, 2013.
50. Hengjun Zhao, Naijun Zhan, Deepak Kapur, and Kim G. Larsen. A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 2012.