A Data Driven Approach for Algebraic Loop Invariants

Rahul Sharma
Stanford University

Saurabh Gupta
UC Berkeley

Bharath Hariharan
UC Berkeley

Alex Aiken
Stanford University

Aditya V. Nori
Microsoft Research India

Abstract

We describe a new algorithm GUESS-AND-CHECK for computing algebraic equation invariants. These invariants are of the form $\wedge_i f_i(x_1, \ldots, x_n) = 0$, where each f_i is a polynomial over the variables $x_1, \ldots, x_n$ of the program. Two novel features of our algorithm are:- (1) it is data driven, that is, invariants are derived from data generated from concrete executions of the program, and (2) it is sound and complete, that is, the algorithm terminates with the correct invariant if it exists.

The GUESS-AND-CHECK algorithm proceeds iteratively in two phases. The "guess" phase uses linear algebra techniques to efficiently derive a candidate invariant from data. This candidate invariant is subsequently validated in a "check" phase. If the check phase fails to validate the candidate invariant, then the proof of invalidity is used to generate more data and this is used to find better candidate invariants in a subsequent guess phase. Therefore, GUESS-AND-CHECK iteratively invokes the guess and check phases until it finds the desired invariant. In terms of implementation, off-the-shelf linear algebra solvers are used to implement the guess phase, and off-the-shelf decision procedures are used to implement the check phase.

We have evaluated our technique on a number of benchmark programs from recent papers on invariant generation. Our results are encouraging – we are able to efficiently compute algebraic invariants in all cases.

Keywords Non-linear, loop invariants, verification

1. Introduction

The task of generating loop invariants lies at the heart of any program verification technique. A wide variety of techniques have been developed for generating linear invariants, including methods based on abstract interpretation [12, 23] and constraint solving [11, 18], among others. The topic has progressed to the point that techniques for discovering linear loop invariants are included in industrial strength tools for software reliability [4, 13, 29].

Recently, researchers have also applied these techniques to the generation of non-linear loop invariants [14, 26, 27, 31–33]. These techniques discover *algebraic invariants* of the form

$$\wedge_i f_i(x_1, \ldots, x_n) = 0$$

where each f_i is a polynomial in the variables $x_1, \ldots, x_n$ of the program.

All these techniques can be classified as either correct-by-construction methods or unsound methods that are based on under-approximate dynamic analyses. The former are based on heavy machinery such as Gröbner bases which ensure that the computed invariants are indeed correct, while the latter are based on efficient techniques for analyzing program executions but unfortunately can report incorrect invariants.

In this paper, we address the problem of invariant generation from a data driven perspective. In particular, we use techniques from linear algebra to analyze data generated from executions of a program in order to efficiently "guess" a candidate invariant. This guessed invariant is subsequently checked for validity via a decision procedure. Furthermore, we are also able to prove that our algorithm always terminates with the desired invariant. Our algorithm GUESS-AND-CHECK for generating algebraic invariants calls these guess and check phases iteratively until it finds the desired invariant – the guess phase is used to compute a candidate invariant I and the check phase is used to validate that I is indeed an invariant. Failure to prove that I is an invariant results in counterexamples or more data which are used to refine the guess in the next iteration.

This guess and check data driven approach for computing invariants has a number of advantages:

- Since the objective of the guess phase is to discover a candidate invariant, it can be made very efficient. Checking whether the candidate invariant is an invariant is done via a decision procedure. Our belief is that using a decision procedure to check the validity of candidate invariant can be much more efficient than using it to infer an actual invariant.
- Since the guess phase operates over data, its complexity is independent of the complexity or size of the program. (The amount of data depends on the number of variables in scope though.) This is in contrast to approaches based on static analysis, and therefore it is at least plausible that a data driven approach may work well even in situations that are difficult for static analysis.

It is also interesting to note that our algorithm is an iterative refinement procedure similar to the counterexample-guided abstraction refinement (CEGAR) [9] technique used in software

```
1:  assume(x=0 && y=0);
2:  while (*) do
3:    writelog(x, y);
4:    y := y+1;
5:    x := x+y;
6:  done
```

Figure 1. Example program 1.

```
1:  assume(s=0 && j=k);
2:  while (j>0) do
3:    writelog(s, i, j, k);
4:    (s,j) := (s+i,j-1)
5:  done
```

Figure 2. Example program 2.

model checking. In CEGAR, we start with an over-approximation of program behaviors and perform iterative refinement until we have either found a proof of correctness or a bug. GUESS-AND-CHECK technique is dual to CEGAR – we start with an under-approximation of program behaviors and add more behaviors until we are done. Most techniques for invariant discovery using CEGAR like techniques have no termination guarantees. Since we focus on the language of polynomial equalities for invariants, we are able to give a termination guarantee for our technique. If a loop has no non-trivial polynomial equation of a given degree as an invariant then our procedure will return the trivial invariant *true*.

Our main contribution is a new sound and complete data driven algorithm for computing algebraic invariants. In particular, our technical contributions are:

- We observe that a known algorithm [27] is a suitable fit for our guessing step. We formally prove that this algorithm computes a sound under-approximation of the algebraic loop invariant. That is, if G is the guess or candidate invariant, and I is an invariant then $G \Rightarrow I$. This guess will contain all algebraic equations constituting the invariants and possibly more spurious equations.
- We precisely define "data-sufficiency", that is, when the test runs are sufficient for our guessing procedure to generate a sound algebraic invariant.
- We augment our guessing procedure with a decision procedure to obtain a sound and (relatively) complete algorithm: if the decision procedure successfully answers the queries made, then the output is an invariant and we do generate all valid invariants up to a given degree d.
- We show that the GUESS-AND-CHECK algorithm terminates after finitely many iterations. We evaluate our technique on benchmark programs from various papers on generation of algebraic loop invariants and our results are very encouraging—GUESS-AND-CHECK terminates on all benchmarks in one iteration, that is, our first guess was an actual invariant. As a result, we were able to obtain invariants using only a few tests.

The remainder of the paper is organized as follows. Section 2 motivates and informally illustrates the GUESS-AND-CHECK algorithm over two example programs. Section 3 introduces the background for the technical material in the paper. Section 4 presents the GUESS-AND-CHECK algorithm and also proves its correctness and termination. Section 6 evaluates our implementation of the GUESS-AND-CHECK algorithm on several benchmarks for algebraic loop invariants. Section 7 surveys related work and, finally, Section 8 concludes the paper.

2. Overview of the Technique

Assume we are given a loop $L = \texttt{while}\ B\ \texttt{do}\ S$ with variables $\vec{x} = x_1, \ldots, x_n$. The initial values of $\vec{x}$, that is, the possible values which $x_1, \ldots, x_n$ can take before the loop starts executing are given by a predicate $\varphi(\vec{x})$. Our goal is to find the strongest $I \equiv \wedge_i f_i(\vec{x}) = 0$, where each f_i is a polynomial defined over the variables $x_1, \ldots, x_n$ of the program such that $\varphi \Rightarrow I$ and $\{I \wedge B\} S \{I\}$. Any I satisfying these two conditions is an invariant for L. If we do not impose the condition that we need the strongest invariant, then the trivial invariant $I = true$ is a valid solution.

Our algorithm has two main phases, the guess phase and the check phase. The guess phase operates over data generated by testing the input program in order to guess a candidate invariant. This candidate invariant is subsequently checked for validity by the check phase which is just a black box call to an off-the-shelf decision procedure. The main insight is that the guess phase is used to quickly guess an invariant which can be checked efficiently by the check phase. Our hope is that this two phased approach is more efficient than a single phase correct-by-construction approach that is the main theme of existing approaches.

We will illustrate our technique over two example programs shown in Figures 1 and 2 respectively. Our objective is to compute loop invariants at the loop head of both these programs. Informally, a loop invariant over-approximates the set of all possible program states that are possible at a loop head.

First, let us consider the program shown in Figure 1. This program has a loop (lines 2 – 6) that is non-deterministic. In line 3, we have instrumentation code that writes the program state (in this case, the values of the variables x and y) to a log file. Equivalently, we could have added multiple instrumentations: after line 1 and before line 6. The loop invariant for this program is $I \equiv y + y^2 = 2x$. Since our approach is data driven, the starting point is to run the program with test inputs and accumulate the resulting data in the log. Assume that we run the program with an input that exercises the loop exactly once. On such an execution, we obtain a single program state $x = 0, y = 0$ at line 3. It turns out that for our technique to work, we need to assume an upper bound d on the degree of the polynomials that constitute the invariant. Note that this assumption can be removed by starting with $d = 0$, and iteratively incrementing d [31]. This assumption also avoids rediscovery of *implied* higher degree invariants. In particular, if $f(\vec{x}) = 0$ is an invariant, then so is the higher degree polynomial $(f(\vec{x}))^2 = 0$. As a consequence, if d is an upper bound on the degree of the polynomial, then we might potentially discover both $f(\vec{x}) = 0$ and $(f(\vec{x}))^2 = 0$ as invariants. For this example, we assume that $d = 2$, which allows us to exhaustively enumerate all the monomials over the program variables up to the chosen degree. For our example, $\vec{\alpha} = \{1, x, y, y^2, x^2, xy\}$ is the set of all monomials over the variables x and y with degree less than or equal to 2.

Using $\vec{\alpha}$ and the program state $x = 0$, $y = 0$, we construct a *data matrix* A which is a 1×6 matrix with one row corresponding to the program state and six columns, one for each monomial in $\vec{\alpha}$. Every entry $(1, j)$ in A represents the value of the j^{th} monomial over the program. Therefore,

$$A = \begin{array}{|c|c|c|c|c|c|} \hline 1 & x & y & y^2 & x^2 & xy \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \quad (1)$$

As we will see in Section 4.1, we can employ the null space of A to compute a candidate invariant I as follows. If $\{b_1, b_2, \ldots, b_k\}$ is a

basis for the null space of the data matrix A, then

$$I \equiv \bigwedge_{i=1}^{k} (b_i^T \begin{bmatrix} 1 \\ x \\ y \\ y^2 \\ x^2 \\ xy \end{bmatrix} = 0) \tag{2}$$

is a candidate invariant that under-approximates the actual invariant. The null space of A is defined by the set of basis vectors

$$\left\{ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\} \tag{3}$$

Therefore, from Equation 2, we have the candidate invariant

$$I \equiv x = 0 \wedge y = 0 \wedge x^2 = 0 \wedge y^2 = 0 \wedge xy = 0 \tag{4}$$

Next, in the check phase, we check whether I as specified by Equation 3 is actually an invariant. Abstractly, if $L \equiv$ `while` B `do` S is a loop, then to check if I is a loop invariant, we need to establish the following conditions:

1. If φ is a precondition at the beginning of L, then $\varphi \Rightarrow I$.
2. Furthermore, executing the loop body S with a state satisfying $I \wedge B$, always results in a state satisfying the invariant I.

The above checks for validating I are performed by an off-the-shelf decision procedure [15]. For our example, we first check whether the precondition at the beginning of the loop implies I:

$$(x = 0 \wedge y = 0) \Rightarrow (x = 0 \wedge y = 0 \wedge x^2 = 0 \wedge y^2 = 0 \wedge xy = 0)$$

This condition is indeed valid, and therefore we check whether the following condition on the loop body (lines 4–5) also holds (we obtain the predicate representing the loop body via symbolic execution [24]):

$$(x = 0 \wedge y = 0 \wedge x^2 = 0 \wedge y^2 = 0 \wedge xy = 0 \wedge y' = y + 1 \wedge x' = x + y') \Rightarrow (x' = 0 \wedge y' = 0 \wedge x'^2 = 0 \wedge y'^2 = 0 \wedge x'y' = 0)$$

This predicate is not valid, and we obtain a counterexample $x' = 1$, $y' = 1$ at line 3 of the program. Let us assume that we generate more program states at line 3 by executing the loop for 4 iterations. As a result, we get a data matrix that also includes the row from the previous data matrix as shown below.

$$A = \begin{array}{|c|c|c|c|c|c|} \hline 1 & x & y & y^2 & x^2 & xy \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 4 & 9 & 36 \\ 1 & 6 & 3 & 9 & 36 & 18 \\ 1 & 10 & 4 & 16 & 100 & 40 \\ \hline \end{array} \tag{5}$$

It is important to observe from A that the monomials x^2 and xy are "quickly growing" monomials. In other words, the values of these monomials increase rapidly with the number of iterations of the loop (heuristically, from our experiments, we find that these rapidly increasing monomials are unlikely to play a part in the invariant). Therefore, we remove them and we show how this is done formally in Section 4.3. After removing rapidly increasing monomials, we obtain the following data matrix.

$$A = \begin{array}{|c|c|c|c|} \hline 1 & x & y & y^2 \\ \hline 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 4 \\ 1 & 6 & 3 & 9 \\ 1 & 10 & 4 & 16 \\ \hline \end{array} \tag{6}$$

As with the earlier iteration, we require the basis of the null space of A and this is defined by the singleton set:

$$\left\{ \begin{bmatrix} 0 \\ 2 \\ -1 \\ -1 \end{bmatrix} \right\} \tag{7}$$

Therefore, from Equation 2 and Equation 7, it follows that the candidate invariant is:

$$I \equiv 2x - y - y^2 = 0 \tag{8}$$

Now, the conditions that must hold for I to be a loop invariant are:

1. $(x = 0 \wedge y = 0) \Rightarrow y + y^2 = 2x$, and
2. $(y + y^2 = 2x \wedge y' = y + 1 \wedge x' = x + y') \Rightarrow (y' + y'^2 = 2x')$

both of which are deemed to be valid by the check phase, and therefore $I \equiv y + y^2 = 2x$ is the desired loop invariant.

Next, consider the program shown in Figure 2, adapted from [33]. For this program, we want to find the loop invariant $I \equiv s = i(k - j)$. Let us assume that the upper bound on the degree of the desired invariant is $d = 2$. Let $\vec{\alpha}$ be the set of all candidate monomials with degree less than or equal to 2. Therefore,

$$\vec{\alpha} = \{1, s, i, j, k, si, sj, sk, ij, ik, jk, s^2, i^2, j^2, k^2\} \tag{9}$$

The number of candidate monomials can be large when compared to the number of monomials which actually occur in the invariant. For instance, if the number of variables is 25, and the upper bound on the degree is 3, then the number of candidate monomials is around 3300. Note that the algorithms for computing null space of a matrix are quite efficient and null space of a matrix with 3300 columns can be computed in seconds. Hence we believe that the guess phase will scale well with the number of variables.

Line 3 appends the program state (in this case, the values of the variables s, i, j and k) to a log file. Running the program with a test input $i = 0$, $j = 1$, $s = 0$ and $k = 1$ results in the following data matrix A.

$$A = \begin{array}{|c|c|c|c|c|c|} \hline 1 & s & i & j & k & \ldots \\ \hline 1 & 0 & 0 & 1 & 1 & \ldots \\ \hline \end{array} \tag{10}$$

The basis for the null space of this matrix is given by the following set of vectors:

$$\left\{ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 1 \\ \vdots \end{bmatrix}, \ldots \right\} \quad (11)$$

This results in the candidate invariant $I \equiv s = 0 \wedge i = 0 \wedge j = 1 \wedge k = 1$. However, the check $(s = 0 \wedge j = k) \implies (s = 0 \wedge i = 0 \wedge j = 1 \wedge k = 1)$ fails and we generate several counterexamples: say $s = 0, j = k$, and $i, k \in \{1, 2, 3, 4, 5\}$. These are 25 counterexamples, and we run some tests from these counterexamples. Two rows of the data matrix A corresponding to these tests are shown below:

1	s	i	j	k	si	sj	sk	ij	ik	jk	...
1	0	3	5	5	0	0	0	15	15	25	...
1	3	3	4	5	9	12	15	12	15	20	...

(12)

By computing the null space, we obtain the candidate invariant $I \equiv s + ij = ik$.

Next, we check whether I is an invariant by checking the following conditions:

$$(s = 0 \wedge j = k) \Rightarrow (s + ij = ik) \quad (13)$$

$$(s + ij = ik \wedge j > 0 \wedge s' = s + i \wedge j' = j - 1) \Rightarrow s' = i(k - j') \quad (14)$$

Both these queries are answered in affirmative, and thus $s = i(k - j)$ is the desired loop invariant. Note that these constraints are quite simple: one possible strategy is to just substitute value of s' and j' in the consequent, in terms of s and j, by using the antecedent. Such simple constraints can be efficiently handled by the recent optimizations in decision procedures for non-linear arithmetic [22]. The cost of solving such constraints can be quite high in general, but we believe that the cost of solving constraints generated from programs occuring in practice is manageable.

In summary, we have informally described how our technique works over two example programs with fairly non-trivial algebraic invariants. Our approach is data driven in that it operates over a data matrix. This has the advantage that our guess procedure is independent of the complexity of the program in contrast to approaches based on static analysis: The guess phase only depends on the number of variables and not on how the variables are used in the program. We have also seen how we can heuristically prune the data matrix in order to avoid processing irrelevant monomials. Finally, our check procedure employs state-of-the-art decision procedures for non-linear arithmetic as a blackbox.

3. Preliminaries

We consider programs belonging to the following language of *while programs*:

$$\mathcal{S} \ ::= \ x{:=}M \mid \mathcal{S}; \mathcal{S} \mid \texttt{if } B \texttt{ then } \mathcal{S} \texttt{ else } \mathcal{S} \texttt{ fi} \mid \texttt{while } B \texttt{ do } \mathcal{S}$$

where x is a variable over a countably infinite sort *loc* of memory locations, M is an expression, and B is a boolean expression. Expressions in this language are either of type `real` or `bool`.

A *monomial* α over the variables $\vec{x} = x_1, \ldots x_n$ is a term of the form $\alpha(\vec{x}) = x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n}$. The *degree* of a monomial is $\sum_{i=1}^{n} k_i$. A *polynomial* $f(x_1, \ldots, x_n)$ defined over n variables $\vec{x} = x_1, \ldots, x_n$ is a weighted sum of monomials and has the following form.

$$f(\vec{x}) = \sum_k w_k x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} = \sum_k w_k \alpha_k \quad (15)$$

where $\alpha_k = x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n}$ is a monomial. We are interested in polynomials over reals, that is, $\forall k \ . \ w_k \in \mathbb{R}$. The *degree* of a polynomial is the maximum degree over its constituent monomials: $max_k \{degree(\alpha_k) \mid w_k \neq 0\}$.

An *algebraic equation* is of the form $f(\vec{x}) = 0$, where f is a polynomial. Given a loop $L = \texttt{while } B \texttt{ do } S$ defined over variables $\vec{x} = x_1, \ldots, x_n$ together with a precondition φ, a *loop invariant* I is the strongest predicate such that $\varphi \Rightarrow I$ and $\{I \wedge B\} S \{I\}$. Any predicate I satisfying these two conditions is an invariant for L. If we do not impose the condition that we need the strongest invariant, then the trivial predicate $I = true$ is a valid invariant. In this paper, we will focus on *algebraic invariants* for a loop. An algebraic invariant $\mathcal{I}$ is of the form $\wedge_i f_i(\vec{x}) = 0$, where each f_i is a polynomial over the variables $\vec{x}$ of the loop.

The next section reviews matrix algebra that is a crucial component of our guess-and-check algorithm. The reader is referred to [21] for an excellent introduction to matrix algebra.

3.1 Matrix Algebra

Let $A \in \mathbb{R}^{m \times n}$ denote a *matrix* with m rows and n columns, with entries from the set of real numbers $\mathbb{R}$. Let $x \in \mathbb{R}^n$ denote a *vector* with n real number entries. Then x denotes an $n \times 1$ column matrix. The vector x can also be represented as a row matrix $x^T \in \mathbb{R}^{1 \times n}$, where T is the matrix transpose operator. In general, the transpose of a matrix A, denoted A^T is obtained by interchanging the rows and columns of A. That is, $\forall A \in \mathbb{R}^{m \times n} . (A^T)_{ij} = A_{ji}$. For example,

$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2 \times 3}$, $x = \begin{bmatrix} 8 \\ 9 \\ 10 \end{bmatrix} \in \mathbb{R}^3$ is a vector, and

$x^T = \begin{bmatrix} 8 & 9 & 10 \end{bmatrix}$, $A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$.

Given two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, their product is the matrix C defined as follows:

$$C = AB \in \mathbb{R}^{m \times p}$$

where the element corresponding to the i^{th} row and j^{th} column $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$. If $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2 \times 3}$, and $B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$, then $C = AB = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$.

The *inner product* of two vectors x and y, denoted $\langle x, y \rangle$, is the product of the matrices corresponding to x^T and y which is defined as:

$$x^T y = \sum_{i=1}^{n} x_i y_i \quad (16)$$

If $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, and $y = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$, then the inner product $\langle x, y \rangle = 32$.

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, their product $y = Ax$ defines a linear combination of the columns of A with coefficients given by x. Alternatively, for a vector $w \in \mathbb{R}^m$, the

product $v = w^T A$ defines a linear combination of the rows of A with coefficients given by the vector w.

A set of vectors $\{x_1, x_2, \ldots, x_n\}$ is *linearly independent* if no vector in this set can be written as a linear combination of the remaining vectors. Conversely, any vector which can be written as a linear combination of the remaining vectors is said to be *linearly dependent*. Specifically, if

$$x_n = \sum_{i=1}^{n-1} \alpha_i x_i \tag{17}$$

for some $\{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}$, then x_n is said to be linearly dependent on $\{x_1, x_2, \ldots, x_{n-1}\}$. Otherwise, it is independent of $\{x_1, x_2, \ldots, x_{n-1}\}$. For example, the set of vectors

$$\left\{ \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}, \begin{bmatrix} 2 \\ 5 \\ 9 \end{bmatrix}, \begin{bmatrix} -3 \\ 9 \\ 3 \end{bmatrix} \right\}$$

is not linearly independent as

$$\begin{bmatrix} -3 \\ 9 \\ 3 \end{bmatrix} = 33 \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} - 18 \begin{bmatrix} 2 \\ 5 \\ 9 \end{bmatrix} \tag{18}$$

The *column rank* of a matrix A is the largest number of columns of A that form a linearly independent set. Analogously, the *row rank* of a matrix A is the largest number of rows of A that form a linearly independent set. It can be easily seen that the column rank of the matrix $B = \begin{bmatrix} 1 & 2 & -3 \\ 3 & 5 & 9 \\ 5 & 9 & 3 \end{bmatrix}$ is 2, and this follows from Equation 18.

From the fundamental theorem of linear algebra [21], we know that for every matrix A, its row rank equals its column rank and is referred to as the *rank* of the matrix A. Therefore, the rank of the matrix B above is equal to 2.

The *span* of a set of vectors $\{x_1, x_2, \ldots, x_n\}$ is the set of all vectors that can be expressed as a linear combination of $\{x_1, x_2, \ldots, x_n\}$. Therefore,

$$span(\{x_1, x_2, \ldots, x_n\}) = \{v \mid v = \sum_{i=1}^{n} \alpha_i x_i, \alpha_i \in \mathbb{R}\} \tag{19}$$

If $\{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathbb{R}^n$ is a set of n linearly independent vectors, then $span(\{x_1, x_2, \ldots, x_n\}) = \mathbb{R}^n$. Thus, any vector $v \in \mathbb{R}^n$ can be written as a linear combination of vectors in the set $\{x_1, x_2, \ldots, x_n\}$. More generally, for any $Q = span(x_1, \ldots, x_n) \subseteq \mathbb{R}^n$, if every vector $v \in Q$ can be written as a linear combination of vectors from a linearly independent set $B = \{b_1, b_2, \ldots, b_k\}$, and B is minimal, then B forms a *basis* of Q, and k is called the *dimension* of Q.

The *range* of a matrix $A \in \mathbb{R}^{m \times n}$ is the span of the columns of A. That is,

$$range(A) = \{v \in \mathbb{R}^m \mid v = Ax, x \in \mathbb{R}^n\} \tag{20}$$

The dimension of $range(A)$ is also equal to $rank(A)$.

The *null space* of a matrix $A \in \mathbb{R}^{m \times n}$ is the set of all vectors that equal to 0 when multiplied by A. More precisely,

$$NullSpace(A) = \{x \in \mathbb{R}^n \mid Ax = 0\} \tag{21}$$

GUESS-AND-CHECK
Input:
– while program L.
– precondition φ.
– bound on the degree d.
Returns: A loop invariant $\mathcal{I}$ for L.

```
1:  x⃗ := vars(L)
2:  Tests := TestGen(φ, L)
3:  logfile := ⟨⟩
4:  repeat
5:     for t⃗ ∈ Tests do
6:        logfile := logfile :: Execute(L, x⃗ = t⃗)
7:     end for
8:     A := DataMatrix(logfile, d)
9:     B := Basis(NullSpace(A)))
10:    if B = ∅ then
11:       //No non-trivial algebraic invariant
12:       return true
13:    end if
14:    I := CandidateInvariant(B)
15:    (done, t⃗) := Check(I)
16:    if ¬done then
17:       Tests := {t⃗}
18:    end if
19: until done
20: return I
```

Figure 3. GUESS-AND-CHECK computes an algebraic invariant for an input while program L with a precondition φ.

The dimension of $NullSpace(A)$ is called its *nullity*. For instance, the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

has a null space determined by the span of the following set of vectors:

$$\left\{ \begin{bmatrix} 2 \\ -3 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ 1 \\ 0 \end{bmatrix} \right\}$$

with $nullity(A) = 2$.

From the fundamental theorem of linear algebra [21], for any matrix $A \in \mathbb{R}^{m \times n}$, we also know the following.

$$rank(A) + nullity(A) = n \tag{22}$$

4. The Guess-and-Check Algorithm

The GUESS-AND-CHECK algorithm is described in Figure 3. The algorithm takes as input a while program L, a precondition φ on the inputs to L, and an upper bound d on the degree of the desired invariant, and returns an algebraic loop invariant $\mathcal{I}$. If $L =$ `while` B `do` S, then recall that $\mathcal{I}$ is the strongest predicate such that

$$\varphi \Rightarrow \mathcal{I} \text{ and } \{\mathcal{I} \wedge B\} S \{\mathcal{I}\} \tag{23}$$

As the name suggests, GUESS-AND-CHECK consists of two phases.

1. *Guess phase* (lines 5–13): this phase processes the data in the form of concrete program states at the loop head to compute

a data matrix, and uses linear algebra techniques to compute a candidate invariant.

2. *Check phase* (line 15): this phase uses an off-the-shelf decision procedure for checking if the candidate invariant computed in the guess phase is indeed a true invariant (using the conditions in Equation 23) [22].

The GUESS-AND-CHECK algorithm works as follows. In line 1, $\vec{x}$ represents the input variables of the while program L. The procedure $TestGen$ is any test generation technique that generates a set of test inputs $Tests$ each of which satisfy the precondition φ. Alternatively, our technique could also employ an existing test suite for $Tests$. The variable *logfile* maintains a sequence of concrete program states at the loop head of L. Line 3 initializes *logfile* to the empty sequence. Lines 4–19 perform the main computation of the algorithm. First, the program L is executed over every test $\vec{t} \in Tests$ via the call to $Execute$ in line 6. $Execute$ returns a sequence of program states (variable to value mapping) at the loop head and this is appended to *logfile*. Note that this can also include the states which violate the loop guard. The function $DataMatrix$ (line 8) constructs a matrix A with one row for every program state in *logfile* and one column for every monomial from the set of all monomials over $\vec{x}$ whose degree is bounded above by d (as informally illustrated in Section 2). The $(i,j)^{th}$ entry of A is the value of the j^{th} monomial evaluated over the program state represented by the i^{th} row.

Next, using off-the-shelf linear algebra solvers, we compute the basis for the null space of A in line 9. As we will see in Section 4.1, Theorem 4.3, the candidate invariant $\mathcal{I}$ can be efficiently computed from A (line 14). If $\mathcal{B}$ is empty, then this means that there is no algebraic equation that the data satisfies and we return *true*. Otherwise, the candidate invariant $\mathcal{I}$ is given to the checking procedure $Check$ in line 15. The procedure $Check$ uses off-the-shelf algebraic techniques [22] to check whether $\mathcal{I}$ satisfies the conditions in Equation 23. If so, then $\mathcal{I}$ is an invariant and the procedure terminates by returning $\mathcal{I}$. Otherwise, $Check$ returns a counter-example in the form of a test input $\vec{t}$ that explains why I is not an invariant – the computation is repeated with this new test input $\vec{t}$, and the process continues until we have found an invariant. Note that when L is executed with $\vec{t}$ then the loop guard might evaluate to *false*. In such a case, the state reaching just before the loop head is added to the log. Hence, the size of *logfile* strictly increases in every iteration.

In summary, the guess and check phases of GUESS-AND-CHECK operate iteratively, and in each iteration if the actual invariant cannot be derived, then the algorithm automatically figures out the reason for this and corrective measures are taken in the form of generating more test inputs (this corresponds to the case where the data generated is insufficient for guessing a sound invariant).

In the next section, we will formally show the correctness of the GUESS-AND-CHECK algorithm – we prove that it is a sound and complete algorithm.

4.1 Connections between Null Spaces and Invariants

In the previous section, we have seen how GUESS-AND-CHECK computes an algebraic invariant over monomials $\vec{\alpha}$ which consists of all monomials over the variables of the input while program with degree bounded above by d. The starting point for proving correctness of GUESS-AND-CHECK is the data matrix A as computed in line 8 of Figure 3. The data matrix A contains one row for every program state in *logfile* and one column for every monomial in $\vec{\alpha}$. Every entry (i,j) in A is the value of the monomial associated with column j over the program state associated with row i.

An invariant $\mathcal{I} \equiv \wedge_i^k (w_i^T \vec{\alpha} = 0)$ has the property that for each

$$w_i, 1 \leq i \leq k, w_i^T a_j = 0 \text{ for each row } a_j \in \mathbb{R}^n \text{ of } A = \begin{bmatrix} a_1^T \\ a_2^T \\ \vdots \\ a_m^T \end{bmatrix}$$

– in other words, $Aw_i = 0$. This shows that each w_i is a vector in the null space of the data matrix A. Conversely, any vector in $NullSpace(A)$ is a reasonable candidate for being a part of an algebraic invariant.

We make the observation that a candidate invariant will be a true invariant if the dimension of the space spanned by the set $\{w_i\}_{1 \leq i \leq k}$ equals $nullity(A)$. We will assume, without loss of generality, that $\{w_i\}_{1 \leq i \leq k}$ is a linearly independent set. Then, by definition, the dimension of the space spanned by $\{w_i\}_{1 \leq i \leq k}$ equals k.

Consider an n-dimensional space where each axis corresponds to a monomial of $\vec{\alpha}$. Then the rows of the matrix A are points in this n-dimensional space. Now assume that $w^T \vec{\alpha} = 0$ is an invariant, that is, $k = 1$. This means that all rows a_j of A satisfy $w^T a_j = 0$. In particular, the points corresponding to the rows of A lie on an $n-1$ dimensional subspace defined by $w^T \vec{\alpha} = 0$. If the data or program states generated by the test inputs $Tests$ (line 2 in Figure 3) is insufficient, then A might not have rows spanning the $n-1$ dimensions. Therefore, from Equation 22, we have $n - rank(A) = nullity(A) \geq 1$ if the invariant is a single algebraic equation. Generalizing this, we can say that $nullity(A)$ is an upper bound on the number of algebraic equations in the invariant. The following lemmas and theorem formalize this intuition.

Lemma 4.1 (Null space under-approximates invariant)**.** *If $\wedge_i^k w_i^T \vec{\alpha} = 0$ is an invariant, and A is the data matrix, then all w_i lie in $NullSpace(A)$.*

Proof. This follows from the fact that for every w_i, $1 \leq i \leq k$, $Aw_i = 0$. □

Therefore, the null space of the data matrix A gives us the subspace in which the invariants lie. In particular, if we arrange the vectors which form the basis for $NullSpace(A)$ as columns in a matrix V, then $range(V)$ defines the space of invariants.

Lemma 4.2. *If $\wedge_{i=1}^k w_i^T \vec{\alpha} = 0$ is an invariant with the set $\{w_1, w_2, \ldots, w_k\}$ forming a linearly independent set, A is the data matrix and $nullity(A) = k$, then $\{w_i \mid 1 \leq i \leq k\}$ is a basis for $NullSpace(A)$.*

Proof. From Lemma 4.1, we know that $w_i \in NullSpace(A)$, $1 \leq i \leq k$. Since $\{w_i \mid 1 \leq i \leq k\}$ is a linearly independent set of cardinality k, it follows that $\{w_i \mid 1 \leq i \leq k\}$ is also a basis for $NullSpace(A)$. □

Theorem 4.3. *If $\wedge_{i=1}^k w_i^T \vec{\alpha} = 0$ is an invariant with the set $\{w_1, w_2, \ldots, w_k\}$ forming a linearly independent set, A is the data matrix and $nullity(A) = k$, then any basis for $NullSpace(A)$ forms an invariant.*

Proof. Let $B = [v_1 \cdots v_k]$ be a matrix with each v_i, $1 \leq i \leq k$ being a column vector, and with $span(\{v_1, \ldots, v_k\})$ equal to $NullSpace(A)$. That is, $\{v_1, \ldots, v_k\}$ is a basis for $NullSpace(A)$. From Lemma 4.1, we know that every w_i, $1 \leq i \leq k$, lies in $span(\{v_1, \ldots, v_k\})$. This means that every w_i, $1 \leq i \leq k$, can be written as follows.

$$w_i = Bu_i \tag{24}$$

for some vector $u_i \in \mathbb{R}^k$. Therefore, if $\wedge_{j=1}^k v_j^T \vec{\alpha} = 0$, from Equation 24, it follows that $w_i^T \vec{\alpha} = 0$, $1 \leq i \leq k$.

From Lemma 4.2, we know that $\{w_1, w_2, \ldots, w_k\}$ form a basis for $NullSpace(A)$, and therefore every v_j, $1 \leq j \leq k$, can be written as a linear combination of vectors from $\{w_1, w_2, \ldots, w_k\}$. From this, it follows that $\wedge_{i=1}^{k} w_i^T \vec{\alpha} = 0 \implies v_j^T \vec{\alpha} = 0$ for all $1 \leq j \leq k$. Thus, $\wedge_{i=1}^{k} w_i^T \vec{\alpha} = 0 \Leftrightarrow \wedge_{j=1}^{k} v_j^T \vec{\alpha} = 0$. □

Theorem 4.3 precisely defines the implementation to the call $CandidateInvariant$ in line 14 of algorithm GUESS-AND-CHECK. Furthermore, Theorem 4.3 also states that we need to have enough data represented by the data matrix A so that $nullity(A)$ equals k, the dimension of the space spanned by $\{w_i\}_{1 \leq i \leq k}$. If this is indeed the case, then $\mathcal{I} \equiv \wedge_{j=1}^{k} w_j^T \vec{\alpha} = 0$ will be an invariant. On the other hand, if the data is not enough, then Lemma 4.1 guarantees that the candidate invariant $\mathcal{I}$ is a sound under-approximation of the loop invariant. If the null space is zero-dimensional, then only the trivial invariant *true* constitutes an invariant over conjunction of polymonial equations that has degree less than equal d.

The question of how much data must be generated in order to attain $nullity(A) = k$ is an empirical one. In our experiments, we were able to generate invariants using a relatively small data matrix for various benchmarks from the literature.

4.2 Check Candidate Invariants

Computing the null space of the data matrix provides us a way for proposing candidate invariants. The candidates are complete; they do not miss any algebraic equations. But they might be unsound. They might contain spurious equations. To obtain soundness, we will use a decision procedure analogous to the technique proposed in [35].

Theorem 4.4 (Soundness)**.** *If the algorithm* GUESS-AND-CHECK *terminates and the underlying decision procedure for checking candidate invariants Check is sound, then it returns an invariant.*

Proof. Using Lemma 4.1, we know that the candidate invariant always under-approximates the true loop invariant. Using the fact that the variable *done* is assigned *true* iff the candidate $\mathcal{I}$ is an invariant, the result is immediate. □

Next, we prove that the algorithm GUESS-AND-CHECK terminates.

Theorem 4.5 (Termination)**.** *If the underlying decision procedure Check is sound and complete, then the algorithm* GUESS-AND-CHECK *will terminate after at most n iterations, where n is the total number of monomials whose degree is bounded by d.*

Proof. Let $A \in \mathbb{R}^{m \times n}$ be the data matrix computed in line 8 in the GUESS-AND-CHECK algorithm. If the candidate invariant $\mathcal{I}$ computed in line 14 of Figure 3 is an invariant (that is, $done = true$), then GUESS-AND-CHECK terminates.

Therefore, let us assume that I is not an invariant, and let $\vec{t}$ be the test or counterexample that violates the candidate invariant as computed in line 15 of the algorithm. As a result, GUESS-AND-CHECK adds $\vec{t}$ to A – call the resulting matrix $\hat{A}$. By construction, we also know that $\vec{t} \notin range(A^T)$. Therefore, it follows that $rank(\hat{A}) = rank(A) + 1$. More generally, adding a counter-example to the data matrix A necessarily increases its rank by 1. From Equation 22, we know that the rank of A is bounded above by n, which implies that GUESS-AND-CHECK will terminate in at most n iterations. □

The next subsection describes a heuristic to remove unnecessary higher degree monomials which results in smaller data matrices, and therefore offers greater scalability.

```
1: (x,y,i) := (1,z,0)
2: assume(s=0 && j=k);
3: while(i<n)
4:   (x,y,i) := (x*z+1,y*z,i+1)
```

Figure 4. Counterexample to the heuristic in Section 4.3.

4.3 Removing higher degree monomials

Consider a situation in which we are interested in the invariant $z = y^3 \wedge x = z^2$, and the bound on the degree of the desired loop invariant is $d = 3$. Since GUESS-AND-CHECK will consider all possible monomials of degree less than or equal to d, it will also consider the monomial $\alpha \equiv x^3$. As a result, we have $\alpha = y^{18}$ and this high degree monomial is unlikely to be a part of the invariant when $d = 3$.

In order to avoid such monomials with an "implicit" high degree, we make a slight modification to the definition of the data matrix A used by the GUESS-AND-CHECK algorithm. We add a column (without loss of generality, let this be the last column) to A representing an additional ghost variable ι. This ghost variable ι plays the role of a loop counter in the while program. Therefore, the value of ι is also part of the program state and is added to *logfile* in line 6 of Figure 3.

Using the value of the ghost variable ι, we can discard monomials which grow at a rate $\omega(\iota^d)$ with the number of loop iterations. It should be noted that this heuristic does not always apply. For instance, consider two variables which grow very quickly with respect to the loop counter but have a linear relationship with each other. Discarding monomials corresponding to these quickly growing variables might result in missing a lower degree invariant. For example, consider the program of Fig. 4 adapted from [26]. This program has the loop invariant `x(z-1)=y-1`. If we let the upper bound on the degree $d = 2$, then `x` and `y` grow at a rate much greater than ι^2. But we cannot discard the monomials formed from `x` and `y` if we want to get the desired invariant. Hence this heuristic can decrese the precision: if the invariant contains a monomial α which we discarded then we will obtain a weaker invariant which does not contain α. Soundness and termination are unaffected.

If this heuristic is applicable, then we will discard the columns of the data matrix A corresponding to the monomials which grow at a rate $\omega(\iota^d)$. Note that it is important to discard these quickly growing monomials from an implementation standpoint as well. For our example, when we execute the loop for generating A, α will take the values y^{18}. This will quickly overflow finite bit numbers and will not allow us to obtain meaningful data.

We will now describe a simple technique for identifying and eliminating quickly growing monomials from the data matrix A. We modify the definition of the data matrix A such that its last column (in this case, the n^{th} column) maintains the value of the ghost loop counter ι for every program state corresponding to a row of A. For every monomial $\alpha_j \in \vec{\alpha}$, we consider the tuples $\{(log(\iota(k)), log(\alpha_j(k))) \mid 1 \leq k \leq m\}$, where $\iota(k)$ denotes the value of ι in k^{th} row or program state in the data matrix A. If we plot these points in $\mathbb{R}^2$, then the slope of the line passing through these points reflects the rate of growth of α_j with respect to the loop counter. For example, if y is a loop counter and $z = y^3$ is a loop invariant, then the monomial $\alpha = z^2$ will produce a line of slope 6. The best fit line passing through a set of points in $\mathbb{R}^2$ can be easily obtained using standard linear algebra techniques [37].

Next, from A, we discard the columns corresponding to the ghost variable and the monomials which give a line of slope more than d in the log-log plot and the rest of the computation proceeds as described by the algorithm GUESS-AND-CHECK.

```
1:  (i,a[0]) = (0,0);
2:  while (i < n) do
3:    i := i+1;
4:    a[i] := a[i-1]+1;
5:  done
6:  assert(a[n] == n);
```

Figure 5. Example with arrays.

5. Discussion

An interesting question is whether the algorithm GUESS-AND-CHECK generalizes to richer theories beyond polynomial arithmetic. This is indeed possible and entails careful design of the representation of data. For instance, if we want to infer invariants in the theory of linear arithmetic and arrays, we can have an additional column in the data matrix for values obtained from arrays. Similarly, we can have a variable which stores the value returned from an uninterpreted function and assign it a column in the data matrix. So it is possible to use our technique to infer conjunction of equalities in richer theories too if we know the constituents of the invariants. This is analogous to invariant generation techniques based on templates [11, 33].

In order to illustrate how the GUESS-AND-CHECK technique would work for programs with arrays, consider the example program shown in Figure 5. We want to prove that the assertion in line 6 holds for all inputs to the program. Assume that we log the values of $a[i]$ and i after every iteration and that the degree bound is $d = 1$. The data matrix that GUESS-AND-CHECK constructs will have two columns and let us assume that we run a single test with input $n = 10$ which results in the data matrix rows corresponding to program states induced by this input at the loop head in the program as shown below.

$$A = \begin{array}{|c|c|}\hline i & a[i] \\ \hline 0 & 0 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \\ 8 & 8 \\ 9 & 9 \\ \hline \end{array} \tag{25}$$

The null space of A is defined by the basis vector $B = [1, -1]^T$, and therefore we obtain the invariant $a[i] = i$ which is sufficient to prove that the assertion holds.

Our approach of using a dynamic analysis technique to generate data in the form of concrete program states and augmenting it with a decision procedure to obtain a sound technique is a general one. Sharma et al. [35] use a similar idea for computing interpolants. We can also take the method for discovering array invariants or polynomial inequalities of [27] and extend it to a sound procedure in a similar fashion.

6. Experimental Evaluation

We evaluate the GUESS-AND-CHECK algorithm on a number of benchmarks from the literature on generation of algebraic invariants [31–33]. All experiments were performed on a 2.5GHz Intel i5 processor system with 4 GB RAM running Ubuntu 10.04 LTS.

Benchmarks The benchmarks over which we evaluated the GUESS-AND-CHECK algorithm are from a number of recent research papers on inferring algebraic invariants. These are shown in the first column of Table 1. The first three programs are benchmarks from [33]. The first program, `Mul2`, multiplies two numbers by repeated addition. The second program, `LCM/GCD`, computes the great common divisor and least common multiple of two numbers. `Div` divides two numbers to obtain a quotient and a remainder. The next two programs are from [31]. `Bezout` uses extended Euclid's algorithm to compute Bezout's coeffecients. `Factor` finds a divisor of a number. The next three programs are examples from [32]. `Prod` is a different way of multiplying two numbers. `Petter` computes the sum $\sum_i i^5$. `Dijkstra` [16] computes least common multiple and greatest commom divisor of two numbers simultaneously. The next eight programs are examples in [30]. `Cubes` computes the cube of a number. The programs `geoReihe1/2/3` compute geometric sums. The program `potSumm1/2/3/4` computes $\sum_i i^x$, where $x \in \{0, 1, 2, 3\}$.

These programs were implemented in C for data generation. Programs such as `potSumm4`, `Petter`, and `geoReihe3` required the removal of higher degree monomials because some of the monomials overflowed C integers (see Setion. 4.3).

Evaluation Our approach can be desribed as a combination of the following components:

1. Test case generation engine: Generally, the programs have tests and we can leverage the existing tests for the purpose of data generation. If these tests are insufficient then the check phase can generate new tests.
2. Program instrumentation: We need to add instrumentation to print the memory state of the program to a log. Standard compiler infrastructures like LLVM [8] can facilitate this task.
3. Linear algebra engine: A variety of engines, like SAGE [36] and MATLAB can compute the null space of a matrix.
4. Loop body to a constraint: This can be considered as a source to source transformation and tools like HAVOC [5] can compile programs written in C to SMT-LIB constraints.
5. Theorem prover: This is an off-the-shelf SMT solver which can reason about non-linear arithmetic [1, 15].
6. Feedback mechanism: We need to extract satisfying assignments from the theorem prover and append them to our data log. This assumes the completeness of the prover: if the formula is SAT the the prover can return a satisfying assignment.

We now describe our implementation and our experimentat results of Table. 1.

The second column of Table 1 shows the number of variables in each benchmark program. The third column shows the upper bound for the degree of the polynomials in the inferred invariant.

The fourth column shows the number of rows of the data matrix. The data or tests were generated naively; each input variable was allowed to take values from 1 to N where N was between 5 and 20 for the experiments. Hence if there were two input variables we had N^2 tests. These tests were executed till termination.

While it is possible to generate tests more intelligently (e.g., by generating counterexamples for failed invariants as suggested in Section 2), using inputs from a very small bounding box demonstrates the generality of our technique by not tying it to any symbolic execution engine. The fifth column shows the number of algebraic equations in the discovered loop invariant. For most of the programs, a single algebraic equation was sufficient. The null space and the basis computations were performed using off-the-shelf linear algebra algorithms in MATLAB. GUESS-AND-CHECK finds the same invariants as reported in the earlier papers [30–33]. For

Name	#vars	deg	Data	#and	Guess time (sec)	Check time (sec)	Total time (sec)
Mul2	4	2	75	1	0.0007	0.010	0.0107
LCM/GCD	6	2	329	1	0.004	0.012	0.016
Div	6	2	343	3	0.454	0.134	0.588
Bezout	8	2	362	5	0.765	0.149	0.914
Factor	5	3	100	1	0.002	0.010	0.012
Prod	5	2	84	1	0.0007	0.011	0.0117
Petter	2	6	10	1	0.0003	0.012	0.0123
Dijkstra	6	2	362	1	0.003	0.015	0.018
Cubes	4	3	31	10	0.014	0.062	0.076
geoReihe1	3	2	25	1	0.0003	0.010	0.0103
geoReihe2	3	2	25	1	0.0004	0.017	0.0174
geoReihe3	4	3	125	1	0.001	0.010	0.011
potSumm1	2	1	5	1	0.0002	0.011	0.0112
potSumm2	2	2	5	1	0.0002	0.009	0.0092
potSumm3	2	3	5	1	0.0002	0.012	0.0122
potSumm4	2	4	10	1	0.0002	0.010	0.0102

Table 1. Name is the name of the benchmark; #vars is the number of variables in the benchmark; deg is the user specified maximum possible degree of the discovered invariant; Data is the number of times the loop under consideration is executed over all tests; #and is the number of algebraic equalities in the discovered invariant; Guess is the time taken by the guess phase of GUESS-AND-CHECK in seconds. Check is the time in seconds taken by the check phase of GUESS-AND-CHECK to verify that the candidate invariant is actually an invariant. The last column represents the total time taken by GUESS-AND-CHECK.

the programs Div and Cubes, the invariants found had some redundancy. For instance, we obtained the following three invariants for the program Div.

$$\begin{aligned} y1 * y3 + y2 * y4 &= y3 * x1 \\ x2 * y3 &= y2 \\ y1 + x2 * y4 &= x1 \end{aligned}$$

These invariants are linearly independent. Using the cancellation laws of multiplication, it can be seen that the third invariant $x1 = y1 + x2 * y4$ follows from the other two. Since these algorithms for computing null spaces operate on raw data, with no knowledge of what the data represents, they will not be able to remove redundancy caused by more complex axioms of arithmetic, such as those corresponding to non-linear operations like multiplications and divisions. The best they can do to remove redundancy is to ensure linear independence. Techniques like Gröbner bases can be used to remove redundancy. Alternatively, a decision procedure can be used to remove redundant algebraic equations [27]. The time (in seconds) taken by the guess phase of GUESS-AND-CHECK is reported in the sixth column of Table 1.

The reader might notice that our implementation is significantly faster when #and equals one. This is due to an implementation trick. MATLAB provides two algorithms for computing the null space of a matrix. The first is a numerical algorithm which has been significantly optimized. The second is an exact algorithm over the rationals which is comparitively slower. Our implementation first tries to use the faster numerical algorithm for computing the null space and the nullity. If the nullity is one, then it is very simple to obtain a vector in the null space which has all its coordinates as exact integers from the numerical output. This process is not so obvious when nullity is more than one. Since we need to supply an exact predicate to the check step, we use the slow exact algorithm when the nullity is more than one.

For example, consider the following matrix:

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

On executing null(A) in MATLAB (the optimized numerical algorithm), we get, within 0.0002 seconds, that the null space is spanned by the vector $[-0.7071, 0.7071]^T$. From this we can conclude that the null space is spanned by the vector $[-1, 1]^T$. On executing null(A,'r') in MATLAB (the slower exact algorithm), we get within 0.0007 seconds that the null space is spanned by $[-1, 1]^T$. Hence using the faster numerical algorithm explains why our implementation is significantly faster when #and is one.

We used Z3 [15] for checking that the proposed invariants were actually invariants (implementation of Check procedure in the GUESS-AND-CHECK algorithm). The theorem prover Z3 implements a new algorithm for deciding satisfiabilty of non-linear arithmetic constraints [22]. It was able to easily handle the simple queries made by GUESS-AND-CHECK. This is because once an invariant has been obtained, the constraint encoding that the invariant is inductive is quite a simple constraint to solve and our naively generated tests were sufficient to generate an actual invariant. For programs with nested loops, we generate candidate invariants at all the loop heads. Then all the invariants are checked. If a counterexample is obtained then then it generates more data and invariant computation is repeated. We continue these guess and check iterations till check phase passes for all the loops. For checking the invariant of outer loop, the inner loop is replaced by its candidate invariant and a constraint is generated. For checking the inner loop, the invariant of the outer loopis used to compute a pre-condition. Finally, the time taken by GUESS-AND-CHECK on these benchmarks is comparable to state-of-the-art correct-by-construction invariant generation techniques [7]. Since these benchmarks are small and time taken by both our technique and [7] is less than a second on these programs, a comparison of run times makes little sense. See Setion. 7 for a more detailed comparison with the previous work.

We leave the collection of a hard benchmark suite for algebraic invariant generation tools as future work.

7. Related Work

We now place the GUESS-AND-CHECK algorithm in the context of existing work on discovering algebraic loop invariants. Sankaranarayanan et al. [33] describe a constraint based technique that uses user-defined templates for computing algebraic invariants. Their objective is to find an instantiation of these templates that satisfies the constraints and this corresponds to an invariant. The constraints that they use contains quantifiers and therefore the cost of solving them is quite high.

Müller-Olm et al. [10, 26] define an abstract interpretation based technique which is sound and generates all possible algebraic invariants which can be obtained by combination of user provided monomials. Rodríguez-Carbonell et al. [31] also describe an abstract interpretation based technique for discovering algebraic invariants. In order to ensure termination, they rely on a widening operator which is precise enough to generate all invariants whose degree is bounded by a user defined constant. They also empirically evaluate their technique which works very well on a number of small benchmark programs. The same authors, in subsequent work [32], under some technical assumptions on program syntax, provide a fully automatic, sound, complete, and terminating algorithm, which was later extended by Kovács [25]..

All of the above mentioned techniques require the heavy technical machinery of Gröbner bases [14] and require restrictions on program syntax, some of which are quite technical, in order to achieve soundness. Recently, Nguyen et al. [27] have proposed a dynamic analysis for inference of candidate invariants. They do not provide any formal characterization of the output of their algorithm and do not prove any soundness or completeness theorems. Their technique has reasonable running time and assumes an upper bound on the degree as input. As noted in [27], our approach can also take advantage of the number of approaches on test case generation developed specifically for the purpose of dynamic invariant generation [19, 20, 38].

More recently, Cachera et al. [7] have extended the work of Müller-Olm et al. [26] to provide an abstract interpretation based algorithm which handles a richer but still somewhat restricted program syntax and achieves a fixpoint in one iteration as opposed to an unknown number of iterations of [26]. Their approach has reasonable running time and requires the degree as input. In contrast, our technique does not require any restriction on program syntax explicitly. We shift the burden of checking an invariant to a decision procedure [22]. In other words, we can handle any program which our decision procedure can handle. This allows us to handle programs with say division and modulo operations, which no existing abstract interpretation based technique is capable of handling: Cachera et al. removed these operators manually [7]. The soundness and completeness of our algorithm follows from the soundness and completeness of the underlying decision procedure. Moreover, together with these theoretical guarantees, our technique has been implemented and evaluated over a number of benchmarks with encouraging results.

Our technique is related to two more pieces of work. Bagnara et al. [3] introduce new variables for monomials and generate linear invariants over them by abstract interpretation. Amato et al. [2] analyze data from program executions to tune their abstract interpretation

The Daikon tool [17] generates likely invariants from tests and templates. Our approach is similar in that it is also based on analyzing data from tests. Furthermore, Daikon does not provide any formal guarantees such as soundness and completeness over the invariants it generates. In the context of Daikon, it is interesting to note from [28] that very few test cases suffice for invariant generation. Indeed, this has been our experience with the GUESS-AND-CHECK as well.

In contrast to all the other techniques which fall into the category of "correct by construction" techniques, the guessing phase of the GUESS-AND-CHECK algorithm will not miss any invariant but can produce spurious ones. The spurious invariants can be removed by running more tests, or in a more systematic fashion by using a decision procedure. Assuming the decision procedure can handle queries which encode the text of the program as constraints, we can get rid of these spurious algebraic equations. Since reasoning about non-linear arithmetic is a hard problem in general, our checking procedure has a high complexity, but as our experiments indicate, the hard cases happen rarely in practice. In contrast to the technique in [33] where the constraint solver solves synthesis constraints for invariant inference, the check phase of our algorithm constructs constraints for verifying whether the given candidate invariant is actually an invariant. These constraints are much simpler than the ones for directly synthesizing invariants which reduces the load on the decision procedure and thus makes the GUESS-AND-CHECK approach very efficient in practice.

Recently, we have proposed another guess-and-check technique for program verification [34]. The proofs computed by this technique can contain arbitrary boolean combinations of linear and non-linear inequalities. However, unlike the approach presented in this paper, the guess phase of the algorithm presented in [34] is neither sound nor complete, and is based on *probably approximately correct* (PAC) learning [6].

8. Conclusion

We have presented the first data driven GUESS-AND-CHECK for discovering algebraic equation invariants that is sound and complete. We use linear algebra techniques to guess an invariant from the data generated from program runs, and use decision procedures for non-linear arithmetic to validate these candidate invariants.

We are also able to formally prove that the guessed invariant is an under-approximation to the actual invariant (soundness), as well as show that GUESS-AND-CHECK will eventually terminate with the correct invariant (completeness). Thus, the key novelty of the GUESS-AND-CHECK approach is the data driven analysis together with formal guarantees of soundness and completeness. We have also informally shown how our approach can be extended to more expressive theories such as arrays.

We have also implemented the GUESS-AND-CHECK algorithm and evaluated it on a number of benchmarks from recent papers on invariant generation and our results are encouraging. Future work includes incorporating the GUESS-AND-CHECK algorithm into a mainstream program verification engine.

References

[1] B. Akbarpour and L. C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3): 175–205, 2010.

[2] G. Amato, M. Parton, and F. Scozzari. Discovering invariants via simple component analysis. *J. Symb. Comput.*, 47(12):1533–1560, 2012.

[3] R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *SAS*, pages 19–34, 2005.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3, 2002.

[5] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *VSTTE*, pages 1–24, 2010.

[6] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, October 1989.

[7] D. Cachera, T. P. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to gröbner bases. In *SAS*, pages 58–74, 2012.

[8] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169, 2000.

[10] M. Colón. Approximating the algebraic relational semantics of imperative programs. In *Static Analysis Symposium (SAS)*, pages 296–311, 2004.

[11] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification (CAV)*, pages 420–432, 2003.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *European Symposium on Programming (ESOP)*, pages 21–30, 2005.

[14] D. A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms - An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate texts in mathematics. Springer, 1997.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[16] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[18] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *Computer Aided Verification (CAV)*, pages 634–640, 2009.

[19] N. Gupta and Z. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *International Conference on Automated Software Engineering (ASE)*, pages 49–58, 2003.

[20] M. Harder, J. Mellen, and M. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering (ICSE)*, pages 60–71, 2003.

[21] K. Hoffman and R. Kunze. *Linear Algebra*. Prentice Hall, second edition, 1971.

[22] D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 339–354, 2012.

[23] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 19(7):385–394, July 1976.

[25] L. Kovács. A complete invariant generation approach for p-solvable loops. In *Ershov Memorial Conference*, pages 242–256, 2009.

[26] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[27] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering (ICSE)*, 2012.

[28] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 229–239, 2002.

[29] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 178–181, 2009.

[30] M. Petter. Berechnung von polynomiellen invarianten. Master's thesis, Fakultät für Informatik, Technische Universität München, 2004.

[31] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[32] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4): 443–476, 2007.

[33] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Principles of Programming Languages (POPL)*, pages 318–329, 2004.

[34] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Program verification as learning geometric concepts. In *submission*, 2012.

[35] R. Sharma, A. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification (CAV)*, pages 71–87, 2012.

[36] W. Stein et al. *Sage Mathematics Software (Version 5.0)*. The Sage Development Team, 2012. `http://www.sagemath.org`.

[37] L. N. Trefethen and D. Bau. *Numerical linear algebra*. SIAM, 1997.

[38] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *International Conference on Automated Software Engineering (ASE)*, pages 40–48, 2003.