Jrl Syst Sci & Complexity (2006) 19: 1–24

A QUANTIFIER-ELIMINATION BASED HEURISTIC FOR AUTOMATICALLY GENERATING INDUCTIVE ASSERTIONS FOR PROGRAMS*

Deepak KAPUR

Received: 8 June 2006
©2006 Springer Science + Business Media, Inc.

Abstract A method using quantifier-elimination is proposed for automatically generating program invariants/inductive assertions. Given a program, inductive assertions, hypothesized as *parameterized* formulas in a theory, are associated with program locations. Parameters in inductive assertions are discovered by generating constraints on parameters by ensuring that an inductive assertion is indeed preserved by all execution paths leading to the associated location of the program. The method can be used to discover loop invariants–properties of variables that remain invariant at the entry of a loop. The parameterized formula can be successively refined by considering execution paths one by one; heuristics can be developed for determining the order in which the paths are considered. Initialization of program variables as well as the precondition and postcondition, if available, can also be used to further refine the hypothesized invariant. The method does not depend on the availability of the precondition and postcondition of a program. Constraints on parameters generated in this way are solved for possible values of parameters. If no solution is possible, this means that an invariant of the hypothesized form is not likely to exist for the loop under the assumptions/approximations made to generate the associated verification condition. Otherwise, if the parametric constraints are solvable, then under certain conditions on methods for generating these constraints, the strongest possible invariant of the hypothesized form can be generated from most general solutions of the parametric constraints. The approach is illustrated using the logical languages of conjunction of polynomial equations as well as Presburger arithmetic for expressing assertions.

Key words Automated software analysis and verification, inductive assertion, loop invariant, quantifier elimination.

1 Introduction

Since Floyd-Hoare-Dijkstra's inductive assertion method, using pre/postconditions and loop invariants, was proposed in the late sixties for verifying properties of imperative programs, discovering invariants of loops automatically has been viewed as a considerable technical challenge. There have been many attempts to address this problem since the pioneering work of [1–3]; the success however has been limited. As a result, research in program verification has suffered

Deepak KAPUR
Department of Computer Science, University of New Mexico, Albuquerque, USA. Email: kapur@cs.unm.edu.
*This research was partially supported by an National Science Foundation (NSF), Information Technology Research (ITR) award CCR-0113611 and an NSF award CCR-0203051.

considerably. Recently, there appears to be a revival of research activities relating to mechanically discovering loop invariants, especially using abstract interpretations and the associated widening operators[4–8].

This paper explores the use of quantifier elimination methods for generating loop invariants/inductive assertions. An inductive assertion is hypothesized to be a parameterized formula in a first-order theory in which certain free variables in a formula are called *parameters*, much in the spirit of the work on deducing subsidiary conditions in the mid 80's on geometry theorem proving by Wu Wen-Tsun[9,10], Chou[11] and the author[12,13].[1] By considering every possible basic path arising from the execution of a given loop, the goal is to generate a verification condition expressed using these parameterized formulas. Constraints on parameters are then generated such that for the values of parameters satisfying these constraints, the verification condition is valid implying that the instantiated formula is indeed an inductive assertion. Once all the basic paths have been considered, the constraints on parameters generated through this process are solved. For every assignment of parameter values for which the associated constraints hold, the instantiated formula is shown to be an inductive assertion.

If a method for generating constraints on parameters from verification conditions (for example, quantifier elimination) is powerful enough that the constraints on parameters produced by the method are logically equivalent to the verification condition obtained by considering all basic cycles of a given loop, then even stronger claims can be made under certain conditions. Assuming that no approximations were made to generate a verification condition, it can be shown that if there is no solution for parameters from the constraints generated (or there is only a trivial solution), then an invariant of the hypothesized form does not exist for the given loop. Further, if there is a finite description of all solutions of constraints on parameters, then the hypothesized formula can be instantiated using this finite description to get the strongest possible invariant (of the hypothesized form) of the loop. Such a claim cannot be made, however, in case of approximations made either in generating a verification condition (e.g., abstracting boolean expressions and tests appearing in a program if they cannot be expressed in the language in which invariants are specified) or an algorithm used to generate constraints on parameters from the verification conditions does not preserve equivalence. Other kinds of approximations using abstract interpretations may be possible as well.

It is shown that parameterized inductive assertions can be specified in Presburger arithmetic, as polynomial equations as well as in Tarski's theory of real closed fields. Each of these theories admits quantifier-elimination; program variables can be eliminated from parameterized formulas expressing verification conditions to generate constraints on parameters. The proposed method is illustrated using examples. We had good experience recently in teaching the method in a course on programming languages.[2] The examples discussed in this paper as well as many other examples can be done easily by hand.

In the next subsection, some recent related work on automatically generating loop invariants is discussed. After discussing some preliminaries in Section 2, the proposed approach is illustrated in detail using an example in Section 3; a polynomial equation is hypothesized to be an loop invariant of a program; after eliminating the program variables from the verification conditions generated from the program, a loop invariant from the program is derived. Section

[1] In my opinion, Wu Wen-Tsun proposed a new perspective for theorem proving by proposing the following questions: Given a finite set of hypotheses, does a conclusion follow? And, if it does not follow, under what conditions on certain variables in the hypotheses and conclusions (which he called *parameters*), does the conclusion follow? He formulated this as a *generic* consequence relation using the concept of genericity in elimination theory and algebraic geometry since the objective was to ensure that a given conjecture is true for all values of variables and almost all values of parameters.

[2] See a brief discussion on this in a presentation given at the *Deduction and Applications* Seminar given at Dagstuhl, Germany, in October 2005 (available at http://www.dagstuhl.de/05431/Materials/).

4 gives the main steps of the proposed approach. Section 5 discusses three logical theories with the desired properties so that formulas in the associated languages can be used for assertions about programs. Section 6 discusses another example illustrating the behavior of the proposed approach if the loop in a program does not have an invariant of the hypothesized form. Section 7 discusses an example that illustrates how the proposed approach can generate multiple independent invariants of a loop in a program. Section 8 illustrates the approach for programs with nested loops. Section 9 considers programs with procedure calls. Section 10 illustrates how invariants expressed using exponentiation can be derived. Section 11 concludes with some remarks for future work. The presentation in the paper is deliberately kept informal with lots of illustration.

1.1 Related Work

In the seventies, Wegbreit and his colleagues[2,14,15], Elpas et al.[1], Manna and Katz[3] proposed writing the semantics of assignment statements in the body of a loop as recurrence equations (also called *difference equations*), finding an explicit expression for the value of each program variable as a function of the number s of loop iterations and then eliminating the variable s to obtain invariant predicates. This is the first explicit use of a quantifier elimination approach for deriving invariants. They were, however, unable to demonstrate the effectiveness of their techniques for nontrivial loop programs.

Karr[16] gave an algorithm for finding *linear equalities* as loop invariants. This work was extended by Cousot and Halbwachs[5], who applied the concept of abstract interpretation[4] to finding *linear inequalities* as invariants. Another extension of Karr's work has been recently proposed by Müller-Olm and Seidl[7], who generate using linear algebra techniques, polynomial equations of bounded degree as invariants in programs with affine assignments. In [17], Colón et al. have used Farakka's lemma and non-linear constraint solving for finding linear inequalities as invariants. This paper is inspired by [17].

Recently, Rodriguez-Carbonell and Kapur[18,19] have proposed a general abstract framework for computing loop invariants using fixed point computation on formulas generated from the body of a given loop. They showed how this framework can be instantiated to give a sound and complete algorithm for computing conjunction of polynomial equations as loop invariants. They demonstrated the use of Gröbner basis algorithms for computing such loop invariants on a family of examples. The main advantage of that approach over the proposed method is that it does not assume any a priori bound on the degree of a polynomial as an invariant. Most interestingly, if there is an invariant of the loop that can be expressed as a conjunction of polynomial equations, their approach will find it without having to know their maximum degree; if such an invariant does not exist, their approach can find that as well, whereas the proposed approach can only determine that the invariant of the hypothesized form does not exist.

The proposed approach is, however, simpler to understand and implement. It is related to an approach in [20]; polynomial invariants whose form is a priori determined (called *templates*) are computed using an extended Gröbner basis algorithm over templates. They also suggested the use of comprehensive Gröbner basis algorithms as well as the theory of real closed fields for quantifier-elimination, but opined that such techniques for constraint solving are intractable even for simple programs. They recommended a collection of weaker techniques under the heading "a practical alternative" and heuristics for solving constraints. A more detailed comparison of their heuristics with the proposed approach is needed since we have succeeded in solving many problems by hand using the proposed approach. Also, the proposed approach gives different results than theirs on many examples, which suggests that our approach is more widely applicable and generates stronger invariants.

2 Preliminaries

We will assume a simple imperative programming language that supports multiple assignment statement, conditional statement, while loop and procedure calls; further, expression evaluation is side-effect free and there are no pointers allowed. For simplicity, we will consider "flat" programs, i.e., programs which do not make any calls to other procedures or functions until Section 9.

An assertion associated with a program point is a formula relating program variables such that whenever control passes through that program point during the execution of the program, the formula is true for the values of the program variables. A loop invariant is an assertion about program variables that holds at the beginning of every iteration of a loop in the program. An inductive assertion associated with a program point is an assertion that follows from other inductive assertions associated with other program points from which the given program point can be reached by an execution of a program fragment. A loop invariant can be shown to follow from an inductive assertion associated with the loop entry requiring that for every possible execution path corresponding to any basic cycle during the execution of the body of the loop, the assertion is preserved by the path. Loop invariant generation is thus typically inductive assertion generation[17]. We will abuse the terminology a bit, and often use loop invariant and inductive assertion interchangeably, although not every invariant is an inductive assertion, but every inductive assertion is an invariant.

For simplicity, we assume that a path in a program is a sequence of assignments, possibly interspersed with boolean tests arising due to conditional statements and loops.

For a given loop $\boldsymbol{L}$, let $p_1, p_2, \cdots, p_k$ be all possible basic cycles[3], consisting of boolean tests and assignment statements, each of which starts at the beginning of the body of the loop and ends at the same point. For a formula $I(\bar{x})$ to serve as an inductive assertion associated with the entry point of the loop, it must be the case that the Hoare triple

$$\{I(\bar{x})\} \quad p_i \quad \{I(\bar{x})\},$$

where p_i stands for the code fragment corresponding to the path p_i, and $\bar{x}$ is the tuple of all of the variables in the program.

Given a multiple assignment statement and the associated Hoare triple, where P, Q are respectively precondition and postcondition of the statement, we have

$$\{P\} \quad \langle x_1, x_2, \cdots, x_l\rangle := \langle e_1, e_2, \cdots, e_l\rangle \quad \{Q\} \qquad \text{if and only if} \quad \left(P \Rightarrow Q|^{\langle e_1, e_2, \cdots, e_l\rangle}_{\langle x_1, x_2, \cdots, x_l\rangle}\right),$$

where $Q|^{\langle e_1, e_2, \cdots, e_l\rangle}_{\langle x_1, x_2, \cdots, x_l\rangle}$ stands for the formula obtained from Q by simultaneously replacing all free occurrences of $x_1, x_2, \cdots, x_l$ by $e_1, e_2, \cdots, e_l$, respectively; x_i's are assumed to be distinct.[4] This is the backward propagation semantics.

3 A Detailed Illustration of the Proposed Approach

In this section, we give an informal overview of the proposed approach for generating loop invariants; no assumption is made about the availability of pre or post conditions surrounding a loop. If they are available, then they can be made use of; otherwise, just the code is used for generating invariants.

[3] A basic cycle of a program is a path in which the beginning point and the end point are the same and the rest of the path is free of any cycles.

[4] If Q is not quantifier-free, one has to be careful in performing substitutions; the free variables appearing in e_i's should not get captured by the scope of quantifiers in Q; to avoid that, quantified variables may need be renamed. For simplicity, we will assume that Q is quantifier-free so that we do not have to worry about this requirement. However, the proposed method extends to quantified inductive assertions as well.

The main idea of the proposed approach is to associate with every loop entry, *parameterized* inductive assertions, each of which is a different quantifier-free formula of the form $I(\bar{x}, \bar{u})$[5] where $\bar{x}$ are the variables standing for program variables which are possibly changing in a program, and $\bar{u}$ are called *parameters* whose values need to be determined for I to serve as an inductive assertion of a given loop. To generate constraints on parameters $\bar{u}$, Hoare triples are generated using I for every possible path of the loop.

Using Floyd-Hoare's inductive assertion method, a verification condition $\Phi(\bar{x}, \bar{u})$ is generated from each program path with the parameters $\bar{u}$ as its only free variables. If $\exists \bar{u}\ \forall \bar{x}\ \Phi$ is valid, it can be shown that there indeed exists an invariant of the hypothesized form $I(\bar{x}, \bar{u})$ with $\bar{u}$ taking specific values; otherwise, such an invariant does not exist. Further, if an equivalent quantifier-free formula, say P, on parameters $\bar{u}$ can be computed from $\forall \bar{x}\ \Phi$, then every assignment of parameter values that makes P true gives an invariant I' obtained from I after instantiating the parameters in it. Further, if a finite description of all assignments making P true can be computed, then the strongest invariant (under certain assumptions) can be generated. This is illustrated below using an example.

3.1 Example

Consider the following simple program to compute the product of two inputs.

function product (M, N: integer) **returns** z: integer
  var x, y: integer **end var**
  $\langle x,\ y,\ z\rangle := \langle M,\ N,\ 0\rangle$;
  while $y \neq 0$ **do**
    if $y \bmod 2 = 1$ **then** $\langle x, y, z\rangle := \langle 2x, (y-1) \operatorname{div} 2, x+z\rangle$;
      else $\langle x, y, z\rangle := \langle 2x, y \operatorname{div} 2, z\rangle$;
    end if
  end while

3.1.1 Generating verification conditions for paths

There are two basic cycles for the loop:

1) p_1 consisting of two tests $y \neq 0$; $y \bmod 2 = 1$; followed by the multiple assignment statement $\langle x, y, z\rangle := \langle 2x, (y-1) \operatorname{div} 2, x+z\rangle$;
2) p_2 consisting of two tests $y \neq 0$; $y \bmod 2 = 0$; followed by the multiple assignment statement $\langle x, y, z\rangle := \langle 2x, y \operatorname{div} 2, z\rangle$.

For a formula $I(x, y, z)$ to be an inductive assertion at the loop entry, it must be the case that

$$\{I(x,y,z) \wedge y \neq 0 \wedge y \bmod 2 = 1\}\ \langle x, y, z\rangle := \langle 2x, (y-1) \operatorname{div} 2, x+z\rangle \{I(x,y,z)\}$$

and

$$\{I(x,y,z) \wedge y \neq 0 \wedge y \bmod 2 = 0\}\ \langle x, y, z\rangle := \langle 2x, y \operatorname{div} 2, z\rangle \{I(x,y,z)\}.$$

Consider the second Hoare triple since its manipulation is easier. Using the backward propagation semantics of the multiple assignment statement, one gets

$$(I(x,y,z) \wedge y \neq 0 \wedge y \bmod 2 = 0) \Rightarrow I(2x, (y \operatorname{div} 2), z),$$

[5]Throughout the paper, we will consider quantifier-free formulas only as candidates for loop invariants in order to keep the presentation simple. We believe that the proposed method can be used to generate arbitrary formulas (with quantifiers) as invariants as well.

where $I(2x, (y \text{ div } 2), z)$ stands for the formula obtained by simultaneously replacing x, y, z by $2x, y \text{ div } 2, z$, respectively in I (this is yet another notation for $I|^{\langle 2x,(y \text{ div } 2),z\rangle}_{\langle x,y,z\rangle}$).

Similarly, from the first Hoare triple, the verification condition is

$$(I(x, y, z) \wedge y \neq 0 \wedge y \bmod 2 = 1) \Rightarrow I(2x, ((y-1) \text{ div } 2), x + z).$$

3.1.2 Hypothesizing *I*

Suppose that $I(x, y, z)$ is a polynomial equation in x, y, z in which the degree of every variable is at most 1. In other words,

$$I(x, y, z) = (A\ xyz + B\ xy + C\ xz + D\ yz + E\ x + F\ y + G\ z + H = 0),$$

where A, B, C, D, E, F, G, H are parameters whose values must be determined, so that the above verification condition is valid.

3.1.3 Generating constraints on parameters from verification conditions

Path 2 Substituting for I in the above formula obtained from the second Hoare triple, gives

$$\begin{aligned}&((A\ xyz + B\ xy + C\ xz + D\ yz + E\ x + F\ y + G\ z + H = 0) \wedge y \neq 0 \wedge y \bmod 2 = 0)\\ \Rightarrow\ &(A\ 2x(y \text{ div } 2)\ z + B\ 2x(y \text{ div } 2) + C\ 2xz + D\ (y \text{ div } 2)z\\ &+ E\ 2x + F\ (y \text{ div } 2) + G\ z + H = 0).\end{aligned}$$

Since $y \bmod 2 = 0$, we can replace y by $2u$ for some u:

$$\begin{aligned}&((A\ 2xuz + B\ 2xu + C\ xz + D\ 2uz + E\ x + F\ 2u + G\ z + H = 0) \wedge u \neq 0)\\ \Rightarrow\ &(A\ 2xuz + B\ 2xu + C\ 2xz + D\ uz + E\ 2x + F\ u + G\ z + H = 0).\end{aligned}$$

The goal is to find values of parameters A, B, C, D, E, F, G, H such that the above formula is valid for all values of x, u, z.[6]

The above formula can easily be simplified to give

$$\begin{aligned}&((2\ Axuz + 2\ Bxu + C\ xz + 2\ Duz + E\ x + 2\ Fu + G\ z + H = 0) \wedge u \neq 0)\\ \Rightarrow\ &(C\ xz - D\ uz + E\ x - F\ u = 0).\end{aligned}$$

The conclusion cannot be simplified any further without introducing any degree 3 terms.

We can use an incomplete heuristic here. One possibility for the above formula to be valid for all values of x, u, z is that the polynomial $(C\ xz - D\ uz + E\ x - F\ u)$ is identically equal to 0; this implies that the coefficient of every term in the polynomial is 0: $C = D = E = F = 0$. This will make $I(x, y, z)$ to be

$$A\ xyz + B\ xy + G\ z + H = 0.$$

There are other possibilities as well for the above formula to be valid but we will not discuss them here.

[6]Strictly speaking, one is concerned for only those values of x, u, z which can possibly arise during the execution of the program. It is not always the case that all values of x, u, z must be considered. This is another approximation made in deriving invariants. It is possible to generate a stronger invariant by considering only reachable states instead of all possible states.

In general, if a theory admits quantifier-elimination, then we need to find a quantifier-free formula equivalent to the above formula universally quantified over x, u, z; this formula is only in A, B, C, D, E, F, G, H.

Path 1 To consider the first basic cycle, it is not necessary to start with the original hypothesized formula. Instead, we can use I to be: $(A\ xyz + B\ xy + G\ z + H = 0)$ after noting that $C = D = E = F = 0$ from the constraints generated while considering the second path. Using the backward semantics of the assignment statement, we get

$$\begin{aligned}&((A\ xyz + B\ xy + G\ z + H = 0) \wedge y \neq 0 \wedge y \bmod 2 = 1)\\ \Rightarrow\ &(A\ 2x((y-1) \text{ div } 2)(x+z) + B\ 2x((y-1) \text{ div } 2) + G\ (x+z) + H = 0).\end{aligned}$$

Since $y \bmod 2 = 1$, replacing y by $2u + 1$ gives

$$\begin{aligned}&(A\ x(2u+1)z + B\ x(2u+1) + G\ z + H = 0)\\ \Rightarrow\ &(A\ 2xu(x+z) + B\ 2xu + G\ (x+z) + H = 0).\end{aligned}$$

This formula can be simplified to give

$$(A\ x(2u+1)z + B\ x(2u+1) + G\ z + H = 0) \Rightarrow (2A\ x^2u - A\ xz + (G-B)\ x = 0).$$

One possibility for the above formula to be valid for all values x, y, z is that if the coefficient of every term in in the polynomial $(2A\ x^2u - A\ xz + (G-B)\ x = 0)$ is 0, implying that $A = 0$ and $G - B = 0$.

After considering both the paths of the loop in the above program: $I(x, y, z) = (B\ xy + B\ z + H = 0)$, this formula can be verified to be an invariant of the above loop.

In a verification condition, if the hypothesis is a conjunction of polynomial equations and the conclusion is a polynomial equation, then quantifier-elimination can be done using *parametric* Gröbner basis construction[21] (also called comprehensive Gröbner basis in [22]). Crudely speaking, the simplification is done by case analysis on the coefficients of the terms appearing in a formula, e.g, whether $A = 0$ or not, $B = 0$ or not, etc; more details can be found in [21,22].

3.1.4 Solving constraints from all paths and initial values to generate invariant

Using the initial values of the variables when the loop is entered, we get an additional constraint on $I(x, u, z)$ above as follows: $I(M, N, 0) = (B\ MN + H = 0)$, which gives $H = -B\ MN$. Substituting for H in I above, we get

$$I(x, y, z) = (B\ xy + B\ z - B\ MN = 0).$$

B can be factored out, thus giving

$$I(x, y, z) = (xy + z - MN = 0)$$

as an invariant of the above loop.[7]

The reader can verify that $I(x, y, z) = (xy + z - MN = 0)$ is indeed an invariant of the loop. Furthermore, it can also be shown that this is the strongest possible invariant expressed as a polynomial equation in which the degree of every variable is at most 1.

If we had hypothesized that $I(x, y, z)$ to be a polynomial in x, y, z in which the degree of every variable was at most 3, even then the above method would have resulted in $xy + z = MN$ as an invariant.

[7] Of course, the case of $B = 0$ makes $I(x, y, z)$ to be a trivial invariant $0 = 0$ (true).

If a precondition and/or a postcondition of a program is available, then it can be used to guess possible hypothesized forms of assertions.

4 An Automatic Method for Generating Invariants

In this section, we present the method illustrated above for automatically generating inductive assertions for imperative programs.

The first step is to fix a logical language to be used for specifying inductive assertions and verification conditions. The language chosen for the above example is the conjunction of polynomial equations. Later, we discuss other languages and associated theories.

Parametric forms of formulas serving as invariants must then be identified. The theory associated with these parametric forms should have the property that the verification condition generated from imperative programs involving such formulas can be solved to get constraints on parameters. As will be clear below, it is not required that the theory associated with parametric forms admit quantifier elimination. If the theory does admit quantifier elimination, then a quantifier elimination procedure can be used to generate conditions on parameters.

To achieve the above, it may become necessary to make some assumptions about the nature of assignment statements and tests allowed. Alternatively, approximations/abstractions will have to be used for assignments and tests so that verification conditions thus generated can indeed be expressed in the chosen logical language. When approximations/abstractions are made, invariants derived using the proposed approach need not be the strongest; further, the proposed approach may miss out in discovering invariants of the hypothesized form.

Given a program consisting of many loops (including nested loops and function/procedure calls), a different assertion, say $I(\bar{x}, \bar{u})$, is attached to the entry of each loop as well as the entry and exit of every function definition (alternatively, every control point in a program can be attached an assertion as in [17], where this labeling is called an *assertion map*). There should be sufficiently many assertions to cover all execution paths of a given program. Here are the steps of the method for mechanically generating inductive assertions.

1) Generate all possible paths from one assertion to another assertion (including itself); a path is either free of cycles or is a basic cycle.
2) For each path, formulate its Hoare triple using assertions associated with the end points of each path.

   Hoare triples so generated could suggest the structure of hypothesized formulas which may need to be used to derive invariants. They could also point out the (in)adequacy of the logical language being used to expressing invariants. For instance, if assignment statements are polynomial expressions in variables, and tests are all arithmetic relations, it is reasonable to use polynomial equations and inequalities for expressing assertions (see, however, examples of exponentiation programs in later sections).

   Derive a verification condition for the loop from the Hoare triples generated from all paths. It may not be possible to generate a verification condition that is a formula in the logical language chosen to specify assertions because boolean conditions serving as tests in conditional statements and loops as well as assignments may not be expressible; for example, boolean tests expressed using $\geq$ are not expressible in a language that has $=$ as the only predicate. In such cases, a verification condition equivalent to the Hoare triple cannot be generated; instead approximate and generate a verification condition expressible in the logical language but weaker than the Hoare triple (i.e., the Hoare triple implies the verification condition).

3) From the verification condition of the form $\forall \bar{x}\ \ \phi(\bar{x}, \bar{u})$, of a loop with $I(\bar{x}, \bar{u})$ as the assertion at its entry, generate a constraint $P(\bar{u})$ on the parameters $\bar{u}$, which is a quantifier-free formula over $\bar{u}$, such that

$$P(\bar{u}) \Rightarrow \forall \bar{x}\ \ \phi(\bar{x}, \bar{u}).$$

For any sequence $\bar{v}$ of parameter values that satisfy $P(\bar{u})$, i.e., $P(\bar{v})$ is true, $\forall \bar{x}\ \ \phi(\bar{x}, \bar{v})$ is also true. In that case, the instantiated assertion $I(\bar{x}, \bar{v})$ with $\bar{v}$ as parameter values is a loop invariant.

If the logical language admits quantifier-elimination, then the result of the quantifier-free elimination method is $P(\bar{u})$ which is equivalent to $\forall \bar{x}\ \phi(\bar{x}, \bar{u})$. The reader should notice that the proposed method does not require a logical language to have a quantifier-elimination method. It is not even necessary for $P(\bar{u})$ to be expressible in the language used for specifying invariants. Furthermore, even if a logical language admits quantifier-elimination, it is not necessary to use it in full strength to derive invariants. Instead, approximations to quantifier elimination suffice, as illustrated below, insofar as it is possible to get a nontrivial $P(\bar{u})$. A complete quantifier-elimination method is however likely to lead to the derivation of stronger inductive assertions.

There can be many distinct value sets for $\bar{u}$ which satisfy P, with each one leading to a possibly distinct invariant. In the trivial case, $P(\bar{u})$ may yield values of $\bar{u}$ which make $I(\bar{x}, \bar{u})$ equivalent to true. In general, $P(\bar{u}) \wedge I(\bar{x}, \bar{u})$ is the strongest invariant obtained using this approach; if the formula can be simplified by solving for $\bar{u}$, we get an equivalent formula expressed using program variables serving as a loop invariant. Otherwise, a finite description of all sets of parameter values satisfying $P(\bar{u})$ needs to be computed. This is illustrated below.

In general, there can be many paths in a loop; this is especially so when loops are nested (see examples in the Appendix). A verification condition is generated from all the paths and then, constraints on parameters are derived. If these constraints are not satisfiable, then either the loop does not have an invariant of the hypothesized form under the assumptions made to derive the verification condition, or a possibly incomplete method used to generate constraints on parameters needs to be improved.

For automatically generating invariants, the computation of generating constraints from a verification condition, however, can be done incrementally. Paths comprising a loop can be considered one at a time (sequentially or in parallel) and constraints on parameters can be incrementally generated corresponding to each path, and then at the end, they can be conjoined. Most importantly, if considering a path leads to constraints on parameters which simplify the hypothesized invariant (as we saw in the example in the previous section, where the consideration of path 2 lead to many parameters becoming 0), then the simplified hypothesized invariant only need to be considered for other paths. Fewer parameters in a verification condition lead to more efficient computation of constraints on parameters.

A major challenge is thus to develop heuristics to use program paths to hypothesize parameterized formulas which are likely candidates for inductive assertions, and the order in which these paths are considered to derive constraints on parameters. It is often useful to hypothesize a highly generic complex formula, identify "good" paths that can be used to prune the formula quickly.

The following properties serve as a basis of correctness of the above method. Since paths considered are sequences of assignment statements interspersed with boolean conditions, it suffices to consider the generation of a verification condition from a Hoare triple involving parameterized formulas and a single assignment (the case of a boolean condition is straightforward). A simple induction on the length of a path establishes these properties for a path as well.

Lemma 1 *Given a parameterized formula $I(\bar{x},\bar{u})$ such that $\forall \bar{x}\ \ \left(I(\bar{x},\bar{u}) \Rightarrow I(\bar{x},\bar{u})^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)}\right)$ is equivalent to $P(\bar{u})$ for some formula P, then for an assignment α of parameter values, $\alpha(P(\bar{u}))$ is true if and only if the instance $J(\bar{x}) = \alpha(I(\bar{x},\bar{u}))$ has the property that*

$$\{J(\bar{x})\}\ (x_1,x_2,\cdots,x_l) := (e_1,e_2,\cdots,e_l)\{J(\bar{x})\}.$$

Proof Assume that $\alpha(P(\bar{u}))$ is true for an assignment α of parameter values. We also have:

$$\forall \bar{x}\ \ \alpha(I(\bar{x},\bar{u})) \Rightarrow \alpha\left(I(\bar{x},\bar{u})^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)}\right)$$

being true. This formula is equivalent to:

$$\forall \bar{x}\ \ \left(J(\bar{x}) \Rightarrow J(\bar{x})^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)}\right),$$

which is the verification condition derived from $\{J(\bar{x})\}\ \ (x_1,x_2,\cdots,x_l) := (e_1,e_2,\cdots,e_l)\{J(\bar{x})\}$.

Suppose there exists an instance of $I(\bar{x},\bar{u})$ obtained by instantiating $\bar{u}$ in it, called $J(\bar{x})$, (i.e. $J(\bar{x}) = \alpha(I(\bar{x},\bar{u}))$) that has the property that $\{J(\bar{x})\}\ \ (x_1,x_2,\cdots,x_l) := (e_1,e_2,\cdots,e_l)\{J(\bar{x})\}$. In other words, we have

$$J(\bar{x}) \Rightarrow J(\bar{x})^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)},$$

which is the same as:

$$\forall\ \bar{x}\ \ \alpha(I(\bar{x},\bar{u})) \Rightarrow \alpha(I(\bar{x},\bar{u}))^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)}.$$

By assumption, this is equivalent to $\alpha(P(\bar{u}))$, implying that $\alpha(P(\bar{u}))$ is true. ∎

In case, $\forall \bar{x}\ \ \left(I \Rightarrow I^{(e_1,e_2,\cdots,e_l)}_{(x_1,x_2,\cdots,x_l)}\right)$ is not equivalent to $P(\bar{u})$ but is implied by $P(\bar{u})$, then for parameters values β that make $P(\bar{u})$ to be false, the instance $J'(\bar{x}) = \beta(I(\bar{x},\bar{u}))$ may or may not satisfy $\{J'(\bar{x})\}\ \ (x_1,x_2,\cdots,x_l) := (e_1,e_2,\cdots,e_l)\ \ \{J'(\bar{x})\}$.

Corollary 1 *Given a constraint P on parameters $\bar{u}$ as defined above in* Lemma 1 *and two parameters assignments α_1 and α_2 making P true, $J(\bar{x}) = \alpha_1(I(\bar{x},\bar{u})) \wedge \alpha_2(I(\bar{x},\bar{u}))$ also satisfies $\{J(\bar{x})\}\ \ (x_1,x_2,\cdots,x_l) := (e_1,e_2,\cdots,e_l)\ \ \{J(\bar{x})\}$.*

5 Admissible Theories

The proposed method requires that a theory used to express an inductive assertion as a parameterized formula should have the following property:

Given a formula $\forall \bar{x}\ \ \phi(\bar{x},\bar{u})$, generate a formula $P(\bar{u})$ such that $P(\bar{u}) \Rightarrow \forall \bar{x}\ \ \phi(\bar{x},\bar{u})$.

It is not necessary for $P(\bar{u})$ to be expressible in the language used for writing inductive assertion. A richer language will be okay also insofar as $P(\bar{u})$ can be used to generate values of parameters which make P true; if an infinite set of parameter values satisfy P, then it will be nice if this set can be succinctly represented.

If the theory has a complete quantifier elimination algorithm for such formulas, i.e., there is an algorithm to generate $\forall \bar{x}\ \phi(\bar{x},\bar{u}) \iff P(\bar{u})$, then stronger claims can be made about the results produced by the proposed method. That is, it can be shown that under certain restrictions on assignment expressions and boolean tests, if there are inductive assertions of the hypothesized form possible for a given program, the proposed approach will find them; furthermore, such assertions will be the strongest possible ones.

Below, we discuss three theories over numbers: (i) Presburger arithmetic in which only linear polynomials related using $=, \leq$ relations (on reals, rationals or integers) can be expressed;

(ii) polynomial equations and inequations with solutions over an algebraic closed field; and (iii) theory of real closed field, which generalizes Presburger arithmetic over the reals and the theory of polynomial equations over the reals to include $=, \leq$ relations on polynomials over the field of reals. For each of these theories, it is possible to find a nontrivial $P(\bar{u})$ satisfying the above property. The theory of real closed fields admits quantifier-elimination. Other theories admit some form of quantifier-elimination (especially a single block of universal quantifiers) if parametric constraints can be expressed in a richer language. Furthermore, many heuristics can be employed to get constraints on parameters from verification conditions.

5.1 Presburger Arithmetic

Consider the theory of Prebsurger arithmetic over the integers with $\leq, =, 0, s, +$.[8] It is well known that this theory admits quantifier-elimination (in the language extended with $\equiv_k$ for every number k); Enderton's book[23] gives one such method. Fourier-Motzkin's algorithm for projection (elimination of variables) as implemented in our theorem prover *Rewrite Rule Laboratory* (*RRL*)[24,25] can be repeatedly used by eliminating a block of existential quantifiers; this is illustrated below.

Without any loss of generality, it can be assumed that the formula from which a variable x has to be eliminated is a conjunction of inequalities of the form $a_i x \leq p_i$ and $q_j \leq b_j x$, $1 \leq i \leq k_1, 1 \leq j \leq k_2$ (equalities can be replaced by inequalities as well).[9] For every pair of inequalities in which x appears with opposite signs, i.e., $ax \leq p$ and $bx \leq q$, where p, q do not have any occurrences of x, a is positive and b is negative, the inequality $0 \leq k_1 p + k_2 q$ is generated, where $lcm(a, b) = k_1 a = k_2 b$. After all such pairs of inequalities have been considered, the resulting inequalities do not have any occurrence of x. In the process, if any inequality of the form $c \leq 0$, where c is positive, is generated, then the formula is unsatisfiable.[10] Otherwise, if the unsatisfiability of a formula is not detected, the resulting formula after elimination of variables may or may not be equivalent to the original formula depending upon whether the eliminated variables are ranging over integers, rationals or reals. In case of integers, it must be ensured that possible intervals where the eliminated variable x ranges over are large enough.

Using the example below, it is shown how Fourier-Motzkin's algorithm can be generalized to do quantifier elimination from parameterized inequalities over the reals, expressed using $0, 1, +, \leq$. For integers, additional conditions must be checked. The reader should have noticed that a parameterized inequality is not expressible in Presburger arithmetic.

Consider the following example taken from [4,5] and also discussed in [17].

var i, j: integer **end var**
$\langle i, j \rangle := \langle 2, 0 \rangle$;
while P_1 **do**
  if P_2 **then** $\langle i, j \rangle := \langle i + 4, j \rangle$;
  else $\langle i, j \rangle := \langle i + 2, j + 1 \rangle$;
  end if
end while

[8] The proposed approach works for Presburger arithmetic over the naturals, rationals as well as the reals.

[9] In the case of rationals or reals, however, strict inequality of the form $a_i x < p_i$ and inequality $a_i x \leq p_i$ have to be distinguished as one cannot be transformed into the other, unlike in the case of naturals and integers. In the case of rationals and reals, the coefficient of x, the variable being eliminated, can be assumed to be 1. Further, if there is an equality relating a variable to other variables, that can be used to eliminate one of the variables. Over the rationals as well as reals, Fourier-Motzkin's algorithm is complete in the sense that linear inequalities are satisfiable if and if no inequality of the form $c \leq 0$, where c is a positive number is generated. In the case of integers, additional tests have to be performed.

[10] It is well-known that Fourier-Motzkin's algorithm can be bad in the worst case. Other algorithms including integer programming or linear programming in case of rationals can be used as well; see, for example, [26].

In the above program, P_1, P_2 are boolean conditions which are assumed not to be expressible as linear equalities and inequalities. If the invariant for the above loop is hypothesized to be a polynomial equation, it can be shown that there does not exist such an nontrivial invariant (e.g., if $I(i,j) = (c_1 i + c_2 j + d = 0)$, then after considering both the paths, I simplifies to $d = 0$; to make I satisfiable, d must be made 0, implying that I is true).

Suppose the invariant $I(i,j)$ is an inequality of the form $c_1 i + c_2 j + d \leq 0$, where c_1, c_2, d are parameters. This leads to the following verification condition which is the conjunction of the following two formulas:

$$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0)$$

and

$$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0).$$

And, from the initial values, the condition is $2c_1 + d \leq 0$.

For this invariant of the above form to exist,

$$\begin{aligned}\Phi = \exists c_1, c_2, d\,[\,(2c_1 + d \leq 0) \wedge (\forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0))\\ \wedge(\forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0))]\end{aligned}$$

needs to be valid over the reals (and hence over the integers), which is indeed the case. Had Φ not been valid, the invariant of the form $c_1 i + c_2 j + d \leq 0$ does not exist.

Let $\Phi'(c_1, c_2, d)$ be such that $\Phi = \exists c_1, c_2, d\ \Phi'(c_1, c_2, d)$. To generate an invariant, values of c_1, c_2, d that make $\Phi'(c_1, c_2, d)$ valid need to be computed. To get the strongest possible invariant of this form, we are interested in finding all possible values of c_1, c_2, d.

The reader should note that the above formula falls outside the language of Presburger arithmetic because of subformulas of the form $c_1 i + c_2 j + d \leq 0$, where c_1, c_2 are not numbers.

Below, we give an obvious generalization of Fourier-Motzkin's algorithm over the reals and rationals to get a quantifier-free formula (without the program variables i, j) equivalent to Φ' with one caveat. The resulting of the generalized Fourier-Motzkin's algorithm is not a quantifier-free formula in Presburger arithmetic as it is expressed in general using $*$. In [27], an elimination method is given for parametric linear constraints over the integers, where bounded quantification is used; the result is a formula outside the language of linear constraints because of its use of $*$ and equivalence modulo $\equiv_e$, where e is an expression. We are developing another approach for solving parametric linear constraints which are likely to result in shorter formulas on parameters; a paper on this is under preparation.

Consider first the subformula

$$\phi(c_1, c_2, d) = \forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0).$$

We will negate ϕ and find a quantifier-free formula equivalent to $\neg\phi$, and then negate the result back to get the quantifier-free formula equivalent to ϕ.

Negating ϕ gives the conjunction of the two literals[11]

$$\neg\phi = (c_1 i + c_2 j + d \leq 0 \wedge -c_1 i - c_2 j - d - 4c_1 < 0).$$

The above pair of inequalities has the coefficient of i to be both c_1 and $-c_1$. Thus, for eliminating i, two cases must be considered:

[11] We are abusing the notation by using i, j to stand for Skolem constants in $\neg\phi$.

1) $c_1 = 0$: now, the inequalities do not have i any more since the above literals become $c_2 j + d \leq 0$ and $-c_2 j - d < 0$. To eliminate j from it, there are additional two subcases:
   (a) $c_2 = 0$: $d \leq 0 \wedge -d < 0$ which is unsatisfiable;
   (b) $c_2 \neq 0$, in which case j can be eliminated by adding the two inequalities giving $0 < 0$ implying this case is not possible either.
2) $c_1 \neq 0$: i can be eliminated by adding the two inequalities giving $-4c_1 < 0$.

This gives the formula

$$[\neg(c_1 = 0 \wedge c_2 = 0) \ \wedge \ \neg(c_1 = 0 \wedge c_2 \neq 0) \ \wedge (c_1 \neq 0 \Rightarrow -4c_1 < 0)],$$

equivalent to $\neg\phi$, which can be simplified. After negating it, we get

$$\phi = (c_1 = 0 \vee c_1 < 0).$$

Similarly, for

$$\forall i, j, ((c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0)),$$

an equivalent formula obtained using the above steps is

$$(c_1 = 0 \wedge c_2 \leq 0) \vee (c_1 \neq 0 \wedge (2c_1 + 2c_2) \leq 0).$$

Combining all these subformulas together, the equivalent formula for Φ above is

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

which simplifies to

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

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.

All solutions of P can be written in terms of the generator set consisting of $\langle c_1 = 0, c_2 = -1, d = 0\rangle$, $\langle c_1 = -1, c_2 = 2, d = 2\rangle$[28]. Corresponding to each generator is an invariant. The conjunction of the invariants corresponding to these generators is $(-j \leq 0 \wedge -i + 2j + 2 \leq 0)$, which can be shown to be the strongest invariant expressed as a conjunction of linear inequalities for the above loop.

As the reader would have noticed, one easy way to generalize Fourier-Motzkin's algorithm for quantifier-elimination of parameterized inequalities over the reals/rationals is to perform case analysis. For a variable x to be eliminated, if the coefficient of x in an inequality is a parametric expression, three cases are considered making the coefficient negative, zero or positive. These constraints can be used to simplify other inequalities in further processing. For an efficient use of constraints and other sophisticated heuristics, see [26]. In [27], this approach is generalized to parametric linear constraints over the integers based on ideas in [29,30].

5.2 Quantifier-free Theory of Conjunctively Closed Polynomial Equations

If assignments in a program are polynomial expressions, its invariants expressed as polynomial equations can be generated using Gröbner basis construction.[12] A typical verification condition in that case is

$$(p_1 = 0 \wedge \cdots \wedge p_k = 0) \Rightarrow q = 0,$$

[12]Since solvability of polynomial equations over the integers is undecidable (Hilbert's 10th problem), solutions are being considered over an algebraically closed field (i.e., complex numbers).

where $p_1, p_2, \cdots, p_k$ as well as q are polynomials in variables $\bar{x}$ and parameters $\bar{u}$.

In [21], a method for computing parametric Gröbner basis from a finite set of parametric polynomials is given; comprehensive Gröbner bases introduced by Weispfenning can also be used[22]. These methods can be adapted to perform quantifier-elimination from parametric formulas of the form

$$(p_1 = 0 \wedge \cdots \wedge p_k = 0) \Rightarrow q = 0,$$

to get an equivalent formula over the parameters, which is a disjunction of conjunction of polynomial equations and inequations over the parameters. Parameters $\bar{u}$ are made lower than variables $\bar{x}$ to be eliminated in term orderings used for computation of Gröbner bases; a block ordering in which the parameters $\bar{u}$ as a block are lower than the variables $\bar{x}$ as a block often works well.

For a conclusion polynomial equation $q = 0$ to follow from the conjunction of polynomial equations, q must be in the radical ideal of the hypothesis polynomials $\{p_1, p_2 \cdots, p_k\}$. It often suffices to compute the Gröbner basis of the hypothesis polynomials, and use it to check whether the conclusion polynomial reduces to zero. In that case, the above formula is valid for all values of parameters; otherwise, if the normal form of the conclusion polynomial q is not 0, then a good heuristic is to require that in the normal form, the coefficient of every term built using $\bar{x}$, the variables to be eliminated, must be identically equal to 0, which gives constraints on parameters. For completeness, it may be necessary to compute the Gröbner basis of the radical ideal of the hypothesis polynomials and use it to reduce the conclusion polynomial.[13] Parametric constraints thus generated can themselves be solved using linear algebra, if they are linear, or using Gröbner basis construction if they are nonlinear.

The construction of a parametric Gröbner basis can be expensive due to branching based on the parametric coefficient of the leading term of a polynomial in a basis. It thus becomes necessary to employ heuristics which can avoid irrelevant/uninteresting paths. The key advantage of the proposed method shows up in this case. Numerous heuristics can be employed to get useful constraints on parameters, in case polynomial equations as invariants exist.

The following heuristic often works very well: (i) Assume that the leading coefficient of each polynomial p_i is nonzero while computing their Gröbner basis (insofar as this can be done consistently); (ii) Once their Gröbner basis is computed, normalize the conclusion polynomial q again assuming the leading coefficient of every polynomial in the Gröbner basis is nonzero; (iii) If the normal form of q is 0, then under the assumptions, the polynomial follows from the hypotheses; otherwise, assert the coefficient of each term in the normal form of q to be 0. Collect all the constraints and solve for them to generate an invariant.

The reader would notice that most of the examples discussed in this paper have been done using the method and heuristics discussed in this section. Even though incomplete, the heuristic has been found to work well on many examples. It is quite efficient in contrast to having to compute a parametric (or comprehensive) Gröbner basis.

5.3 Theory of Real Closed Fields

The theory of real closed fields, whose decidability was shown by Tarski in 1930's, is the most expressive theory for specifying parametric polynomial formulas as inductive assertions. In this theory, both $\leq$ as well as $=$ can be used as relations over polynomial expressions. Further, solutions are sought over the field of real numbers, unlike the treatment in the previous subsection where the solutions are over an algebraically closed field (e.g., the field of complex numbers).

[13] For a related approach based on refutational completeness for proving Euclidean geometry theorems as well as for deducing subsidiary conditions as well as missing hypotheses of geometry problem formulations, see [13].

The theory of real closed field admits quantifier-elimination. Decision procedures for the theory has been extensively studied and implemented over the last 40 years; see [31]. Specialized implementation of decision procedures along with the ones for low degree polynomial constraints are available–software packages REDLOG[32] and QEPCAD[33] implemented in computer algebra systems REDUCE and Maple, respectively, as well as a Mathematica implementation are particularly promising. Unlike in the cases of the theory of polynomial equations and Presburger arithmetic, nothing special has to be done to consider parametric formulas since even parameterized formulas are in the theory; it suffices to make a distinction between variables and parameters. Finding constraints on parameters thus amounts to eliminating universally quantified variables (standing for program variables) from the verification condition generated from considering all execution paths of a given loop; this can be done in parallel for all paths or incrementally as illustrated above.

The main drawback of the decision procedures for the theory of real closed field is that they work only on toy problems in practice. That is unlike other methods for Presburger arithmetic as well as for polynomial equations over an algebraically closed field, which work reasonably well on many interesting problems, even though all of these methods are the same order of complexity (doubly exponential in the number of variables). An interested reader may consult [34] for more details.

For the examples discussed in this paper, our attempts to use the above-mentioned software packages implementing decision procedures for the theory of real closed field have been quite discouraging. These implementations took a long time to produce complex incomprehensible outputs or ran out of memory space. We observed two issues. Because the number of variables to be eliminated from verification conditions can be at least three or four, theses softwares are not able to do quantifier-elimination on formulas arising even for simple examples.

Second, special heuristics need to be investigated that can exploit the special structure of the formulas serving as verification conditions. There is a close structural relationship between the hypotheses and the conclusions since assignment statements are replacing variables by polynomial expressions.

6 Hypothesizing an Invariant of Unsuitable Form

We show how the proposed method works in case a formula that cannot serve as an invariant is hypothesized to be an invariant since this is not known a priori.

Consider again the example introduced in Section 3. In contrast to a polynomial equation in which every variable has at most degree 1, suppose $I(x, y, z)$ is hypothesized as a linear equation in x, y, z. As we shall see below, the proposed method would discover that no linear equation can serve as an invariant.

Let $I(x, y, z) = (A\ x + B\ y + C\ z + D = 0)$.

Considering the path p_2, the verification condition generated is

$$\begin{aligned} &(A\ x + B\ y + C\ z + D = 0 \wedge y \neq 0 \wedge y \bmod 2 = 0) \\ \Rightarrow\ &(2\ Ax + B\ (y \text{ div } 2) + C\ z + D = 0). \end{aligned}$$

Of course, the goal is to see whether there exist values of A, B, C, D such that for all values of x, y, z, the above formula is valid.

After substituting $2u$ for y, the conclusion above becomes: $2A\ x + B\ u + C\ z + D = 0$. The above implication is not valid if both $A \neq 0$ and $B \neq 0$. The formula equivalent to it, in terms of A, B, C, D, is

$$(A = 0 \wedge B = 0) \vee (A \neq 0 \wedge B = 0 \wedge C = 0 \wedge D = 0)\ \vee (A = 0 \wedge B \neq 0 \wedge C = 0 \wedge D = 0).$$

It can be easily verified that for the values of parameters satisfying these constraints, the above formula is indeed preserved by path p_2. The only mildly interesting case is $A = 0, B = 0$, as other cases lead to the invariant being trivial. With $A = 0$, $B = 0$, we get $Cz + D = 0$.

Considering the path p_1 leads to $C = 0$, again giving the trivial invariant.

Considering the path p_1 on the original $I(x, y, z)$ and replacing y by $2u + 1$:

$$\begin{aligned} &(A\ x + 2B\ u + B + C\ z + D = 0 \wedge y \neq 0 \wedge y \bmod 2 = 1) \\ \Rightarrow\ &(2\ Ax + B\ u + C\ x + C\ z + D = 0). \end{aligned}$$

One gets $A = -C \wedge B = 0$ or $A = 0 \wedge C = 0 \wedge B = D$ as possible constraints along with $(A = 0 \wedge B = 0 \wedge C = 0) \vee (A \neq 0 \wedge B = 0 \wedge C = 0 \wedge D = 0)$. However, each case reduces to the trivial invariant.

Since we did not make any approximations in this analysis, it implies that the loop in the program does not have a linear equation as an invariant as hypothesizing an invariant of this form trivializes to true.

As this example illustrates, the proposed approach will not generate an incorrect invariant.

7 Conjunction of Formulas as an Invariant: Multiple Solutions for Constraints on Parameters

Constraints on parameters generated from different execution paths in a loop may not, in general, lead to the parameters getting totally determined. Instead, the parameter constraints can have multiple solutions. Each solution to parameter constraints can be shown to lead to an invariant; some of these invariants are independent of each other as the following example illustrates. The proposed method in such a case produces a conjunction of formulas as an invariant, in which each formula corresponds to an independent solution of parameter constraints. Under certain conditions, such a conjunction of formulas can be shown to be the strongest possible invariant of the loop of the hypothesized form. The example below illustrates this as well as what happens if different parametric forms are hypothesized as assertions of the same program. This example also illustrates that an invariant in general need not be an inductive assertion.

Consider the following simple loop for computing the floor of the square root of a natural number.

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

There is only one path in this loop consisting of the loop test and the assignment statement:

$$s \leq N; \quad \langle a,\ s,\ t\rangle := \langle a + 1,\ s + t + 2,\ t + 2\rangle.$$

7.1 Linear Equation as an Invariant

Let us hypothesize the invariant $I(a, s, t)$ to be a linear polynomial equation of the form: $A\ a + B\ s + C\ t + D = 0$. The verification condition generated is

$$(A\ a + B\ s + C\ t + D = 0 \wedge s \leq N) \Rightarrow A\ (a + 1) + B\ (s + t + 2) + C\ (t + 2) + D = 0.$$

Simplifying it leads to

$$(A\ a + B\ s + C\ t + D = 0 \wedge s \leq N) \Rightarrow A + B\ (t + 2) + 2C = 0.$$

This formula is valid if $B\ t + (A + 2B + 2C) = 0$ for any t, implying that $B = 0 \wedge A = -2C$. So the invariant becomes $-2C\ a + C\ t + D = 0$.

Using the initial values, we get $C + D = 0$ implying that $D = -C$.

The invariant gets further refined to $-2C\ a + C\ t - C = 0$. After removing C as a factor gives $-2a + t - 1 = 0$ as the invariant. It can be proved to be the strongest linear equation invariant of the above loop.

However, this invariant is not strong enough to capture fully the semantics of the body of the loop. Particularly, it is not sufficient to prove the post condition of the above program, namely, the result a is the largest natural number such that $a^2 \leq N$; we need additional invariants.[14]

7.2 Nonlinear Polynomial Equation as Invariant

Suppose the invariant is hypothesized to be a polynomial equation of degree 2, that is,

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

where $u_1, u_2, \cdots, u_{10}$ are parameters. After simplifying the verification condition, the following constraints on parameters are generated. Each of u_2, u_4, u_6 becomes 0, and

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

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. Values of the dependent parameters are then determined.

1) Assigning $u_{10} = 1$, gives $u_7 = 2, u_9 = -1$; all other parameters are 0; the polynomial $2a - t + 1 = 0$ is an invariant; recall that this is also the invariant generated when I is hypothesized earlier to be a linear polynomial equation.
2) Assigning $u_5 = 1$ gives $u_1 = u_7 = u_8 = -1, u_9 = 1$; all other parameters are 0; this gives $-a^2 + at - a - s + t = 0$ as another invariant.
3) Assigning $u_3 = 1$ gives $u_7 = -2, u_8 = -4, u_9 = 3$; all other parameters are 0; one gets the polynomial $t^2 - 2a - 4s + 3t = 0$ as the third invariant.

Each of these ~~independent~~ invariants is an inductive assertion of the loop as well.

Consider the conjunction of the invariants corresponding to all independent solutions of the above constraint set. This formula can be shown to be the strongest invariant of the above loop, expressed in the polynomial equation form in which the degree of every variable is at most 2. Every other such invariant can be obtained from these three invariants; this is so because every solution of the above constraint set can be expressed in terms of the above independent solutions.

Let us analyze this from another but an equivalent perspective. If we had plugged in solutions of parameters from the above constraints into the above hypothesized invariant, we would get

$$\begin{aligned} I(a, s, t) = &-u_5\ a^2 + u_3\ t^2 + u_5\ at + (-2u_3 - u_5 + 2u_{10})a + (-4u_3 - u_5)s \\ &+(3u_3 + u_5 - u_{10})t + u_{10} = 0. \end{aligned}$$

[14] This is an example of a program where having access to a post condition would have helped to guess the hypothesized form of an invariant, particularly, it should be a polynomial relating a^2 and s, since after the loop is exited, the negation of the loop condition, $s > N$, is true.

If the coefficient of each parameter is collected,

$$I(a,s,t) = u_5\ (-a^2 + at - a - s + t) + u_3\ (t^2 - 2a - 4s + 3t) + u_{10}(2a - t + 1) = 0.$$

This gives the three invariants.

In general, if the constraints on parameters generated from all paths are linear, then a finite conjunction of polynomial equations corresponding to each independent solution can be shown to be the strongest invariant of the hypothesized form (assuming no approximations were made while deriving verification conditions and simplifying them to get constraints on parameters).

For the above example, $s = (a+1)^2$ is also a loop invariant; it can be expressed as a linear combination of $t = 2a + 1$ and $a^2 = at - a - s + t$. However, it can be proved not to be an inductive assertion of the loop.

8 Nested Loops

All the examples so far have been of simple loops. However, the approach is general and it works for programs with nested loops as well as with procedure calls. In general, for every control construct, it suffices to associate an inductive assertion each, with every entry and exit of the control construct. For nested loops, it often suffices to associate an inductive assertion at every loop entry. Below, we use a simple example to illustrate this.

function Square(T: integer) **returns** s: integer
  var m, n, s, x: integer **end var**
  $\langle m,\ n,\ s,\ x\rangle$:=$\langle 0,\ 0,\ 0, T\rangle$;
  while $x \neq 0$ **do**
    $\langle x,\ n,\ m\rangle$:=$\langle x-1,\ m+2,\ m+1\rangle$;
    while $m \neq 0$ **do**
      $\langle s,\ m\rangle$:=$\langle s+1,\ m-1\rangle$;
    end while
    m:=n;
  end while

At each loop entry, an invariant is associated; for the outer-loop, let it be $I(m,n,s,x)$, and for the inner loop, let it be $J(m,n,s,x)$. Corresponding to various paths in the program, the following verification conditions are generated:

$$(I(m,n,s,x) \wedge x \neq 0) \Rightarrow J(m,n,s,x)|^{\langle x-1, m+2, m+1\rangle}_{\langle x,n,m\rangle}$$

for the path from the outer loop entry to the inner loop entry. The path corresponding to the body of the inner loop gives

$$(J(m,n,s,x) \wedge m \neq 0) \Rightarrow J(m,n,s,x)|^{\langle s+1, m-1\rangle}_{\langle s,m\rangle},$$

and

$$(J(m,n,s,x) \wedge m = 0) \Rightarrow I(m,n,s,x)|^{n}_{m}$$

is the verification condition corresponding to the path from the exiting of the inner loop to the beginning of the outer loop body.

From the initial values, we have $I(0,0,0,T)$.

From the above verification conditions, it seems reasonable to speculate invariants to be linear equations since each variable in the assignment is getting incremented or decremented by a constant value.

Suppose $I(m,n,s,x)$ and $J(m,n,s,x)$ are hypothesized to be parametric linear equations, e.g., $I(m,n,s,x) : A\ m{+}B\ n{+}C\ s{+}D\ x{+}E = 0$ and $J(m,n,s,x) : F\ m{+}G\ n{+}H\ s{+}K\ x{+}L = 0$. Simplifying using the techniques discussed above, $I(m,n,s,x)$ and $J(m,n,s,x)$ can be derived, respectively, to be $2x - 2T + m = 0$ and $2x - 2T + n = 0$. These are indeed invariants of the respective loops.

On the other hand, if $I(m,n,s,x)$ and $J(m,n,s,x)$ are hypothesized to include terms in which variables can have degree ≤ 2, we get

$$I(m,n,s,x) : \{2x - 2T + m = 0,\quad s + x^2 - T^2 + m\ x = 0,\quad m^2 - n^2 = 0\}$$

and

$$J(m,n,s,x) : \{2x - 2T + n = 0,\quad m + s + x^2 + s\ n - T^2 = 0\}.$$

Notice that both I and J are conjunction of formulas; each includes the equation obtained when I, J are hypothesized to be linear equations.

Other examples of programs with nested loops are given in the Appendix and are briefly reviewed later in Section 10.

9 Procedure Calls

The proposed approach works for programs which call other functions/procedures. As in the case of nested loops (or for that matter, any control construct), inductive assertions are associated with the entry and exit of a procedure. Replacing of formal parameters of a procedure/function by actual values is done by a sequence of assignment statements. This is illustrated using a variation of the example from the previous section where the innermost loop is made into a procedure call.

function Square1(T: integer) **returns** s: integer
  var m, n, s, x: integer **end var**
  $\langle m,\ n,\ s,\ x\rangle{:=}\langle 0,\ 0,\ 0, T\rangle$;
  while $x \neq 0$ **do**
    $\langle x,\ n\rangle{:=}\langle x - 1,\ m + 2\rangle$;
    $t := s$;
    $s :=$ **func**$(t, m + 1)$;
    $m{:=}n$;
  end while

The above program calls the following procedure/function which has a loop, whose invariant is $t{+}m = t_0{+}m_0$, where t_0, m_0 are the values of the formal parameters with which the function is called. If the procedure terminates, then the result $u = t = t_0 + m_0$, since m becomes 0.

function func(t, m: integer) **returns** u: integer
  while $m \neq 0$ **do**
    $\langle t,\ m\rangle{:=}\langle t + 1,\ m - 1\rangle$;
  end while
  $u := t$

Using this specification of **func**, invariants for the loop in **Square1** can be deduced to be $m + 2x - T = 0, s + x^2 - T^2 + mx = 0$.

It may be possible to generate invariants/inductive assertions in a bottom-up fashion; in other words, inductive assertions for functions/procedures are first generated and then they are used to generate inductive assertions for locations in programs using these functions/procedures.

However, stronger inductive assertions about these functions/procedures may be needed than those already derived; in that case, it may become necessary to derive such stronger invariants, depending upon how they are used. Thus interaction between the context where a function/procedure is used/called and its code become critical.

9.1 Recursive Procedures

We illustrate below how recursive procedures can be handled in similar fashion; this is done using a slight variation of the above example, where **func** is changed to a recursive procedure instead of its body being the loop.

function func(t, m: integer) **returns** u: integer
  if $m = 0$ **then** $u := t$
  else $u :=$ **func**$(t + 1, m - 1)$;

Let $I(t, m, u)$: $At + Bm + Cu + D = 0$ be a hypothesized invariant associated with the above recursive procedure. The two paths in the body of the recursive procedure lead to the following verification condition:

$$(At + Bm + Cu + D = 0 \wedge m = 0) \Longrightarrow (At + Bm + Ct + D = 0) \wedge$$
$$(At + Bm + Cu + D = 0 \wedge m \neq 0) \Longrightarrow (A(t + 1) + B(m - 1) + Cu + D = 0).$$

From the second conjunction, we get the constraint $B = A$. From the first conjunction, the only interesting constraints are $C = -A \wedge D = 0$. Putting these together gives $I(t, m, u)$: $At + Am - Au = 0$; factoring out A, we have $u = m + t$ as an invariant.

10 Inductive Assertions Expressed Using Exponentiation

The language of polynomial equalities and inequalities is restricted for writing inductive assertions even for programs operating on numbers. Consider the following program for computing exponentiation, which uses the same strategy as the above program in Section 3.1 for computing multiplication.

function exp (M, N: integer) **returns** z: integer
  var x, y: integer **end var**
  $\langle x,\ y,\ z\rangle$:=$\langle M,\ N,\ 1\rangle$;
  while $y \neq 0$ **do**
    if $y \bmod 2 = 1$ **then** $\langle x, y, z\rangle$:=$\langle x * x, (y - 1) \text{ div } 2, x * z\rangle$
      else $\langle x, y, z\rangle$:=$\langle x * x, y \text{ div } 2, z\rangle$;
    end if
  end while

As the reader would note, this program has no meaningful invariant expressed in the language of polynomials. Instead, we need a language which allows exponentiation function. The above method can still be used to derive invariants if log function is introduced in the logical language; then, $\log(x^y)$ is equivalent to $y \log(x)$ which is a polynomial in y and $\log(x)$. Using this trick, we have successfully derived invariants of the above as well as many other programs.

For the above program, we can hypothesize an inductive assertion to be associated with the loop entry to be a polynomial in which the degree of every variable (including their logs) is at most 1. Using that hypothesized form, we can obtain an invariant to be: $z * (x^y) = M^N$, which using log becomes $\log(z) + y * \log(x) = N * \log(M)$, a polynomial equation in $x, y, z, \log(x), \log(y), \log(z)$.

The above exponentiation program can be written in terms of + as a nested loop in many different ways. Appendix includes two such variations.

The derivation of invariants for the programs in the Appendix using the proposed approach and using log function is still possible, but it is now necessary to take log of polynomials in program variables. In principle, that is doable by introducing auxiliary variables which stand for such polynomial expressions. For the first program in the Appendix, the invariant for the outer-loop is $x * u = M^{N+1-y}$, whereas the invariant for the inner loop is $x * u + z = M^{N+1-y}$. Using log, the invariant of the inner loop is $\log(x * u + z) = (N + 1 - y) * \log(M)$. For the second program in the Appendix, the invariant for the outer-loop is $z * x^y = M^N$, whereas the invariants for the two inner-loops are $(u * v + z) * x^y = M^N$ and $(u * v + x)^y * z = M^N$.

As the reader would notice, these invariants, however, are not necessarily easy to guess. Guessing parameterized forms of inductive assertions from a program description and quickly dropping/adding terms from hypothesized forms based on feasible paths in a program are interesting challenges, which are worth exploring.

11 Concluding Remarks and Future Work

We have presented an approach for automatically generating inductive assertions of imperative programs using heuristics based on quantifier-elimination techniques. From these inductive assertions, invariants associated with loops in a program can be derived. The approach first generates verification conditions that inductive assertions must satisfy to capture the effects of program statements in an execution path. Parameterized formulas are used for expressing inductive assertions. Constraints on parameters are generated such that each such set of parameters values satisfying these constraints leads to a specific inductive assertion. This can be done using quantifier-elimination methods and related heuristics for elimination program variables from verification conditions.

If the parameter values satisfying these constraints can be succinctly described, such a description can be used to generate a general inductive assertion.

We have illustrated the approach using programs for which assertions can be expressed in three related theories – (i) linear inequalities over the reals, rationals and integers, (ii) polynomial equalities and their negations over complex numbers, and finally, (iii) polynomial inequalities and equalities over the reals. Quantified variables can be eliminated from parameterized formulas expressed in these theories to get constraints on parameters (sometimes, using functions outside the language). Under certain conditions on programs and based on the power of quantifier-elimination techniques, our approach can generate the strongest possible invariants of pre-determined forms. It can also deduce if invariants of such forms do not exist for a given program.

The main advantage of the approach is that many example programs can be handled by hand. We have successfully used this approach in a course on programming to teach students about loop invariants.

The approach is currently being implemented and its effectiveness needs to be further investigated. It will be worth exploring how quantifier-elimination techniques can exploit the special structure of the verification conditions generated from programs. There is also a need to explore (incomplete) heuristics for simplifying a verification condition for eliminating variables and obtaining useful constraints on parameters.

A potentially interesting topic of research is to generate hypothesized forms of inductive assertions based on the nature of assignment statements appearing in a program. One promising idea is to first generate the constraints on an inductive assertion caused by the program state-

ments on execution paths, and then speculate on the possible forms of the inductive assertion.

Most importantly, there is a need to investigate richer logical languages in which properties of complex data structures, including arrays, records, etc. can be expressed; the structure of hypothesized forms of formulas serving as inductive assertions will need to be explored as well.

12 Acknowledgments

The author would like to thank Stan Lee for posing a question in the author's seminar course on program verification and theorem proving offered at the UNM Department of Computer Science in Spring/Fall 1999. This led the author once again to think about invariant generation as a quantifier elimination problem. The author is especially thankful to Sushmita Roy who in the author's course on programming language systems in Fall 2005, implemented a preliminary version of the method as well as tried many problems for which polynomials cannot be used as invariants. Thanks also to Enric Rodríguez-Carbonell for collaboration on the topic of automatic generation of invariants and many discussions on the topic of this paper.

References

[1] B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger, *Research in Interactive Program-Proving Techniques*, Stanford Research Institute, Menlo Park, California, USA, May 1972.

[2] S. German and B. Wegbreit, A synthesizer of inductive assertions, *IEEE Transactions on Software Engineering*, 1975, **1**(1): 68–75.

[3] S. Katz and Z. Manna, Logical analysis of programs, *Communications of the ACM*, 1976, **19**(4): 188–206.

[4] 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*, Los Angeles, California, ACM Press, New York, NY, 1977, 238–252.

[5] P. Cousot and N. Halbwachs, Automatic discovery of linear restraints among variables of a program, *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Tucson, Arizona, ACM Press, New York, NY, 1978, 84–97.

[6] M. Müller-Olm and H. Seidl, Polynomial constants are decidable, in 9*th Static Analysis Symposium* (SAS) (LNCS 2477), Springer-Verlag, 2002.

[7] M. Müller-Olm and H. Seidl, Precise interprocedural analysis through linear algebra, in *Symposium on Principles of Programming Languages*, 2004, 330–341.

[8] E. Rodríguez-Carbonell and D. Kapur, An abstract interpretation approach for automatic generation of polynomial invariants, in 11*th Symposium on Static Analysis* (SAS) (LNCS 3148), Verona, Italy, Springer-Verlag, August 2004, 280–295.

[9] W. Wu, On the decision problem and the mechanization of theorem proving in elementary geometry, *Scientia Sinica*, 1978, **21**: 150–172.

[10] W. Wu, Basic principles of mechanical theorem proving in geometries, *J. of Automated Reasoning*, 1986, **2**: 221–252.

[11] S. C. Chou, *Mechanical Geometry Theorem Proving*, D. Reidel Publishing Company, Dordrecht, Netherlands, 1988.

[12] D. Kapur, Geometry theorem proving using Hilbert's Nullstellensatz, in *Proc.* 1986 *Symposium on Symbolic and Algebraic Computation* (SYMSAC 86), 1986, 202–208.

[13] D. Kapur, A refutational approach to Geometry theorem proving, *Artificial Intelligence*, 1988, **37**: 61–93.

[14] B. Wegbreit, The synthesis of loop predicates, *Communications of the ACM*, 1974, **17**(2): 102–112.

[15] B. Wegbreit, Property extraction in well-founded property sets, *IEEE Transactions on Software Engineering*, 1975, **1**(3): 270–285.

[16] M. Karr, Affine relationships among variables of a program, *Acta Informatica*, 1976, **6**: 133–151.
[17] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, Linear invariant generation using non-linear constraint solving, in *Computer-Aided Verification* (CAV 2003), Lecture Notes in Computer Science, Springer Verlag, 2003, **2725**: 420–432.
[18] E. Rodríguez-Carbonell and D. Kapur, Automatic generation of polynomial loop invariants: Algebraic foundations, in *Intl. Symp. on Symbolic and Algebraic Computation* (ISSAC), Santander, Spain, July 2004, 266–273.
[19] E. Rodríguez-Carbonell and D. Kapur, Automatic generation of polynomial loop invariants for imperative programs, in *Intl. Colloquium on Theoretical Aspects of Computing* (ICTAC 2004) (LNCS 3407), China, Springer-Verlag, Feb. 2005, 325–340.
[20] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, Non-linear loop invariant generation using Gröbner Bases, in *Symp. on Principles of Programming Languages*, 2004, 318–329.
[21] D. Kapur, An approach for solving systems of parametric polynomial equations, in *Principles and Practices of Constraint Programming* (ed. by Saraswat and Van Hentenryck), MIT Press, 1995, 217–244.
[22] V. Weispfenning, A Comprehensive Gröbner Basis Algorithm, *J. of Symbolic Computation*, July 1992, **14**: 1–29.
[23] H. Enderton, *Mathematical Logic, An Introduction*, Academic Press, 1992.
[24] D. Kapur and X. Nie, Reasoning about numbers in Tecton, in *Proceedings 8th International Symposium on Methodologies for Intelligent Systems* (ISMIS'94), October 1994, 57–70.
[25] D. Kapur and H. Zhang, An overview of rewrite rule laboratory (RRL), *Journal of Computer and Mathematics with Applications*, 1995, **29**: 91–114.
[26] R. Loos and V. Weispfenning, Applying linear quantifier elimination, *Computer Journal*, 1993, **36**(5): 450–462.
[27] A. Lasaruk and T. Sturm, Weak quantifier elimination for the full linear theory of the integers – A uniform generalization of Presburger arithmetic, Technical Report, MIP-0604, University of Passau, April 2006.
[28] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, 1998.
[29] V. Weispfenning, Complexity and uniformity of elimination in Presburger arithmetic, in *Intl. Symp. on Symbolic and Algebraic Computation* (ISSAC), Maui, US, July 1997, 48–53.
[30] V. Weispfenning, The complexity of almost linear Diophantine problems, *Journal of Symbolic Computation*, 1990, **10**: 395–403.
[31] G. Collins and H. Hong, Partial cylindrical algebraic decomposition for quantifier elimination, *Journal of Symbolic Computation*, 1991, **12**(3): 299–328.
[32] A. Doltzmann, and T. Sturm, REDLOG: Computer algebra meets computer logic, MIP-9603, Technical Report, University of Passau, Sep. 1996.
[33] H. Hong, et al., http://www.cs.usna.edu/ qepcad/B/WhatisQEPCAD.html.
[34] A. Doltzmann, T. Sturm, and V. Weispfenning, Real quantifier elimination in practice, Technical Report, MIP-9720, University of Passau, Dec. 1997.

Appendix: Some Additional Examples of Programs with Nested Loops

```
function exp1(M, N: integer) returns z: integer
    var x, y, z, u: integer end var
    ⟨x, y, z, u⟩:=⟨M, N, 0, 1⟩;
    while y ≠ 0 do
        z := 0;
        while u ≠ 0 do
            if y mod 2 = 1 then ⟨x, u, z⟩:=⟨x + x, (u − 1) div 2, x + z⟩
                           else ⟨x, u, z⟩:=⟨x + x, u div 2, z⟩;
            end if
        end while
        ⟨x, y, u⟩:=⟨M, y − 1, z⟩;
    end while

function exp2(M, N: integer) returns z: integer
    var x, y, z, u: integer end var
    ⟨x, y, z, u, v⟩:=⟨M, N, 0, 1, 1⟩;
    while y ≠ 0 do
        if y mod 2 = 1 then
            {⟨y, u, v, z⟩:=⟨y − 1, x, z, 0⟩;
              while u ≠ 0 do
                  if u mod 2 = 1 then ⟨v, u, z⟩:=⟨v + v, (u − 1) div 2, z + v⟩
                                 else ⟨v, u, z⟩:=⟨v + v, u div 2, z⟩;
                  end if
              end while}
                else
            {⟨y, u, v, x⟩:=⟨y div 2, x, x, 0⟩;
              while u ≠ 0 do
                  if u mod 2 = 1 then ⟨v, u, x⟩:=⟨v + v, (u − 1) div 2, x + v⟩
                                 else ⟨v, u, x⟩:=⟨v + v, u div 2, x⟩;
              end if
              end while}
    end while
```