Using Symbolic Summation and Polynomial Algebra for Imperative Program Verification in *Theorema* [1]

Laura Kovács, Tudor Jebelean [a] and Deepak Kapur [b]

[a] *Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, Institute e-Austria, Timişoara, Romania*

[b] *Department of Computer Science, School of Engineering, University of New Mexico*

Abstract

An approach utilizing combinatorics, algebraic methods and logic is presented for generating polynomial loop invariants for a family of imperative programs operating on numbers. The approach has been implemented in the *Theorema* system, which seems ideal for such an integration given that it is built on top of the computer algebra system *Mathematica*, has a theorem prover for first-order logic as well as for mechanizing induction. These invariant assertions are then used for generating the necessary verification conditions as first-order logical formulae, based on Hoare logic and the weakest precondition strategy. The approach has been successfully tried on many programs implementing interesting number theoretic algorithms. It is also shown that for a subfamily of loops, called P-solvable loops, the approach is complete in generating polynomial equations as invariants.

Key words:
program verification, invariant generation, symbolic summation, Gröbner Bases

Email addresses:
`{kovacs,jebelean}@risc.uni-linz.ac.at` (Laura Kovács, Tudor Jebelean),
`kapur@cs.unm.edu` (Deepak Kapur).

[1] The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project. The *Theorema* system is supported by FWF (Austrian National Science Foundation) - SFB project F1302.

1 Introduction

Loop invariants are the key to deductive verification of programs. Automatically checking and finding invariants and intermediate assertions are crucial in the analysis and verification of sequential programs. This paper discusses an approach for automatically generating polynomial equations as loop invariants in the *Theorema* system [2]. Exploiting the symbolic manipulation capabilities of the computer algebra system *Mathematica* on which *Theorema* resides, it is possible to use several techniques. These include solving recurrence relations and manipulating polynomial relations, and most importantly, combine them with automated methods for theorem proving in first-order predicate calculus, domain specific reasoning, as well as induction theorem proving since they are supported in *Theorema*.

A family of loops, called *P-solvable* loops, has been identified, for which the value of each program variable can be expressed as a polynomial in terms of the initial values of variables (when the loop is entered), loop counter, and some new variables that are polynomially related. It is shown that for such loops, polynomial equations as loop invariants can be automatically generated. Further, if the body of such loops includes assignments and conditionals, then the approach generates a complete set of polynomial equations as invariants from which any polynomial equation serving as an invariant can be derived. Many nontrivial number-theoretic algorithms can be shown to be implemented using P-solvable loops.

The approach has been implemented in *Theorema* and successfully attempted on a number of programs. We have also implemented a simple imperative programming environment in *Theorema*, and thus we are able to integrate in the system the verification of procedural programs, by using a *verification condition generator* (VCG) based on Hoare logic [14] and the weakest precondition method [10,6]. Since the creative part of imperative program verification is the "guessing" of loop invariants and termination terms (ranking functions), the proposed approach makes *Theorema* more amenable to imperative program verification.

The key steps of the proposed approach are: (i) assignment statements from a loop body are extracted which are used to generate a system of recurrence equations describing the behavior of the loop's variables that are changed at each iteration; (ii) methods from algebraic combinatorics implemented in *Mathematica* [28,19] are used to exactly solve the recurrence equations, thus producing a closed form (i.e. solutions that are functions of the loop counter) for each loop variable; these closed forms however may be expressed in terms of exponentials (which are loop counters) of algebraic numbers; (iii) algebraic dependencies among exponentials of algebraic numbers occurring in the closed forms of the loop variables are derived using algebraic and combinatorial methods implemented in *Mathematica* [20]. The result of these steps is that every program variable can be expressed as a polynomial in terms of the initial values of variables (when the loop is entered), loop counter, and some new variables that are polynomially related. Loop counters are then eliminated using Gröbner basis algorithm [1] to derive a finite set of polynomial identities among the program variables as invariants. From this finite set, any polynomial identity serving a loop invariant can be derived.

The obtained invariants, together with the user-asserted non-polynomial invariant properties, are used further in the verification process for generating automatically the necessary verification conditions, and to prove them by the available *Theorema* provers. Polynomial identities found by an automatic analysis are useful for program verification, as they provide non-trivial valid assertions about the program, and thus significantly simplify the verification task. Finding valid polynomial identities (i.e. invariants) has applications in many classical data flow analysis problem [26], e.g. constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

We have tested the proposed approach on a number of examples in imperative program verification [23]; some of these example are presented in this paper.

The current paper extends earlier papers [24,25] by a correctness proof of the invariant generation algorithm for P-solvable loops with conditionals.

The rest of the paper is organized as follows: Section 2 gives a brief overview on related work for invariant generation, followed by section 3 containing the presentation of some theoretical notions that are used further in the paper. In section 4 we present our method for polynomial invariant generation, and illustrate the algorithm on a concrete example in section 5, presenting also the imperative verification and programming environment in *Theorema*. Section 6 concludes with some ideas for the future work.

2 Related Work

Research into methods for automatically generating loop invariants goes a long way, starting with the works [11,17]. However, success was somewhat limited for cases where only few arithmetic operations (mainly additions) among program variables were involved. Recently, due to the increased computing power of hardware, as well as advances in methods for symbolic manipulation and automated theorem proving, the problem of automated invariant generation is once again getting considerable attention. Particularly, using the abstract interpretation framework [4], many researchers [27,29,30,16] have proposed methods for computing automatically polynomial invariant identities using polynomial ideal theoretic algorithms.

In [27,30], the invariant generation problem is translated to a constraint solving problem. In [30], non-linear (algebraic) invariants are proposed as templates with parameters; constrains on parameters are generated (by forward propagation) and solved using the theory of ideals over polynomial rings. In [27], backward propagation is performed for non-linear programs (programs with non-linear assignments) without branch conditions, by computing a polynomial ideal that represents the weakest precondition for the validity of a *generic polynomial relation* at the target program point. Both approaches need to fix a priori the degree of a generic polynomial template being considered as an invariant.

A related approach for polynomial invariant generation without any a priori bound on the degree of polynomials is presented in [29]. It is observed that polynomial invariants constitute an ideal. Thus, the problem of finding all polynomial invariants reduces to computing a finite basis of the associated polynomial invariant ideal. This ideal is approximated using a fix-point procedure by computing iteratively the Gröbner bases of a certain polynomial ideal. The fixed point procedure is shown to terminate when the list of (conditional) assignments present in the loop constitutes a *solvable mapping.* In [16], a method for invariant generation using *quantifier-elimination* [3,8] is proposed. A *parameterized* invariant formula at any given control point is hypothesized and constraints on parameters are generated by considering all paths through that control point. Solutions of these constraints on parameters are then used to substitute for parameters in a parameterized invariant formula to generate invariants.

In our work we do not need to fix a priori the degree of a polynomial assertion, and do not use the abstract interpretation framework either. Instead, recurrence relations expressing the value of each program variable at the end of any iteration are formulated and solved exactly. Structural conditions are imposed on recurrence relations so that their closed form solutions can be obtained by advanced symbolic summation techniques. Since these closed form expressions can involve exponentials of algebraic numbers, algebraic dependencies among these exponentials need to be identified which can be done automatically. Finally, for eliminating the loop counter from these closed form solutions expressed as polynomials, a Gröbner basis computation is performed; however, unlike, [29], we do not need to perform Gröbner bases computations multiple times, but only once. The proposed approach is very much in the spirit of the earlier works [7,31,18] from the 70's.

3 Theoretical Preliminaries

This section contains a collection of some well-known definitions and facts about linear recurrences, ideals and algebraic dependencies, which are needed later on. For additional details see [5,9,20].

Throughout this paper we assume that $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{Q}$, $\mathbb{R}$, etc.), and by $\bar{\mathbb{K}}$ we denote its algebraic closure. All rings are commutative.

Recurrences.

Definition 3.1 *Gosper-summable recurrences [12].*
A Gosper-summable recurrence f(n) *in $\mathbb{K}$ is a recurrence of the form:*

$$f(n+1) = f(n) + h(n+1)\ (n \geq 1), \tag{1}$$

where $h(n)$ is a hypergeometric term in $\mathbb{K}$, e.g. $h(n)$ can be a product of factorials, binomials, pochhammers, rational-function terms and exponential expressions in the summation variable n (all these factors can be raised to an integer power).

The closed-form solution of a Gosper-summable recurrence can be exactly computed [12]; for doing so, we use the recurrence solving package `zb`, implemented in *Mathematica* by the RISC Combinatorics group [28].

Example 3.1 *Given the Gosper-summable recurrence,*

$$x(n+1) = x(n) + n * 2^n,\ \ n \geq 0$$

with inital value $x(0)$, we obtain its closed form:

$$x(n) = x(0) + 2 + 2^{1+n}(n-1).$$

Definition 3.2 *C-finite recurrences [32,9].*
A C-finite recurrence f(n) *in $\mathbb{K}$ is a (homogeneous) linear recurrence with constant coefficients, i.e. it is of the form:*

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \ldots + a_{r-1} f(n+r-1)\ (n \geq 1), \tag{2}$$

where r is the order *of the recurrence, and $a_0, \ldots, a_{r-1}$ are constants from $\mathbb{K}$, with $a_0 \neq 0$.*

By writing x^i for each $f(n+i)$, $i = 0, \ldots, r$, the corresponding characteristic polynomial $c(x)$ *of f(n) is:*

$$c(x) = x^r - a_0 - a_1 x - \cdots - a_{r-1} x^{r-1}. \tag{3}$$

A crucial and elementary fact about C-finite recurrences is that they always admit a closed form solution (i.e. a solution that is expressed, without recursion, as a function of the recurrence index) [9].

Proposition 3.1 *Closed Form of C-finite Sequences.*
The closed form of a C-finite sequence $f(n)$ in $\mathbb{K}$ is:

$$f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n, \tag{4}$$

where $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ are the distinct roots of the characteristic polynomial of $f(n)$, and $p_i(n)$ is a polynomial in n whose degree is less than the multiplicity of the root θ_i $(i = 1, \ldots, s)$.

For obtaining the closed-form solutions of C-finite recurrences we use the `SumCracker` package, a *Mathematica* implementation by the RISC Combinatorics group [19].

Ideals. Algebraic Dependencies.

Definition 3.3 *Ideals.*
Let R be a ring. A set $\mathbf{A} \subseteq R$ is called an ideal *in R iff for all $a_1, a_2, a \in \mathbf{A}$, $p \in R$ we have $a_1 + a_2 \in \mathbf{A}$ and $pa \in \mathbf{A}$. We write $\mathbf{A} \trianglelefteq R$ to denote that $\mathbf{A}$ is an ideal in R.*

For $p_1, \ldots, p_r \in R$ we denote by $\langle p_1, \ldots, p_r \rangle$ the smallest ideal containing $p_1, \ldots, p_r$. If $\mathbf{A} = \langle p_1, \ldots, p_r \rangle$, we say that $\mathbf{A}$ is generated *by $p_1, \ldots, p_r$ and that $p_1, \ldots, p_r$ is a* basis *of $\mathbf{A}$.*

It is necessary for our work that we can effectively compute with ideals. This is possible by using Buchberger's algorithm for Gröbner basis computation [1]. A Gröbner basis is a basis for an ideal that has special properties that make it possible to answer algorithmically questions about ideals, such as ideal membership of a polynomial, equality and inclusion of ideals, etc. A detailed presentation of the Gröbner bases theory can be found in [1].

Definition 3.4 *Algebraic Dependencies among Exponential Sequences.*
Let $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ be algebraic numbers, and their corresponding exponential sequences $\theta_1^n, \ldots, \theta_s^n \in \bar{\mathbb{K}}$.
An algebraic dependency *(or* algebraic relation*) [20] of these sequences is a polynomial p such that*

$$p(\theta_1^n, \ldots, \theta_s^n) = 0, \quad (\forall n \geq 1). \tag{5}$$

Using results from [20], if the exponential sequences of the algebraic numbers $\theta_1, \ldots, \theta_s$ are related algebraically, all their algebraic dependencies can be obtained automatically. For doing so, first the set of all integer tuples

$L = \{(m_1, \ldots, m_s) \in \mathbb{Z}^s : \prod_{i=1}^{s} \theta_i^{m_i} = 1\}$ is determined exhaustively, and then, by computing the ideal $\left\langle \left\{ \prod_{i=1}^{s} x_i^{a_i} - \prod_{i=1}^{s} x_i^{b_i} : a \in \mathbb{N}^s, b \in \mathbb{N}^s, a - b \in L \right\} \right\rangle$, the algebraic dependencies among the exponential sequences is determined.
Example 3.1.

- The algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 4$ is: $\theta_1^{2n} - \theta_2^n = 0$.
- There is no algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 3$.

For automatically determining all algebraic dependencies among exponential sequences, we use the `Dependencies` package [20] implemented in *Mathematica* by the RISC combinatorics group.

4 Generation of Invariant Polynomial Identities

The algorithm for polynomial invariant generation presented in this paper combines computer algebra and algorithmic combinatorics, in such a way that the end of the invariant generation process all valid polynomial assertions of a *P-solvable loop* are automatically obtained. The definition of P-solvable loops is available in our earlier conference papers [24,25].

Informally, an imperative loop is a **P-solvable imperative loop** (to stand for polynomial-solvable) iff the closed form solution of the loop variables are linear combinations of geometric sequences $\theta^n \in \bar{\mathbb{K}}$ with polynomial coefficients $p(n) \in \mathbb{K}$ (where $n \in \mathbb{N}$ is the loop counter and the coefficients of $p(n)$ are determined by the initial values of the loop variables), with the property that there exist algebraic dependencies among all θ^n.

The class of P-solvable loops includes the simple situation when the expressions in the assignment statements are affine mappings. Moreover, as presented in equation (4), any closed form solution of a loop variable defined by a C-finite recurrence is a linear combinations of terms of the form $p_i(n)\theta_i^n$. Hence, by imposing the restriction of having algebraic dependencies among the exponential sequences θ_i^n, a loop with only C-finite assignments is P-solvable.

Our experience shows that most practical examples operating on numbers exhibit the P-solvable loop property, hence the class of P-solvable loops covers at least a significant part of practical programming.

Remark 4.1 *At the current stage of our work, we consider loops with the property that their assignment statements are either Gosper-summable, geometric series, C-finite recurrences or can be handled by the technique of gener-*

ating functions [13]. For solving these recurrences we use the mentioned Mathematica *packages, implemented by the RISC Combinatorics group [28,19,20].*

4.1 P-solvable Loops with Assignments Only

We present briefly our algorithm that finds all polynomial invariants for P-solvable loops with assignments only. We denote by n the loop counter, and by $x_1, \dots, x_m$ ($m > 1$) the recursively changed loop variables. More details can be found in [24,25].

Polynomial Invariant Generation for P-solvable Loops with Assignments only

(1) **Solving recurrences.**
  (a) Extract the recurrence equations of the loop variables from the body of the P-solvable loop;
  (b) By recurrence solving, obtain the system of closed forms of $x_1, \dots, x_m$ as polynomials in terms of initial values of variables, loop counter and new variables representing exponential sequences of θ_j ($j = 1, \dots, s$);
  (c) Introduce the notations: $y_0 = n, y_1 = \theta_1^n, \dots, y_s = \theta_s^n$, and determine the algebraic dependencies $\mathbf{A}$ among $y_0, \dots, y_s$. Thus

$$x_i(n) = q_i(y_0, y_1, \dots, y_s), \qquad i = 1, \dots, m \tag{6}$$

  where $q_i \in \mathbb{K}[y_0, y_1, \dots, y_s]$, having their coefficients determined by the initial values of x_i.

(2) **Polynomial invariant generation.**
The generators of the ideal of polynomial relations among $x_1, \dots, x_m$ are computed from $\mathbf{A}$ and (6) by elimination of $y_0, \dots, y_s$ using Gröbner basis w.r.t. a suitable elimination order.
Thus, our algorithm finds the generators of the ideal of polynomial relations from which any polynomial invariant can be derived.

The restrictions at the various steps of the algorithm are crucial, namely in case that the recurrences cannot be solved exactly, or their closed forms do not fulfill the P-solvable form, our algorithm fails in generating valid polynomial relations among the loop variables.

4.2 P-solvable Loops with Conditionals and Assignments

We consider a generalization of the invariant generation algorithm from the previous section, namely we present an invariant generation algorithm of P-solvable loops with conditionals. The key idea is to do first program transformation (see Prop. 4.1), namely transform P-solvable loops with conditionals (i.e. outer loops) into nested P-solvable loops with assignments only (i.e. inner loops), and then apply the algorithm presented in the previous section to obtain all polynomial invariants of the inner loops.

Proposition 4.1 *Transformation Rule of Loops with Conditionals.*
The P-solvable loop with conditionals and assignments

$$\{I\} \quad \texttt{While}[b, c_1; \texttt{If}[b_1 \texttt{ Then } c_2 \texttt{ Else } c_3]; c_4] \quad \{I \wedge \neg b\} \tag{7}$$

is equivalent with the P-solvable loop with nested P-solvable loops containing only assignments:

$$\{I\} \qquad \begin{array}{l} \texttt{While}[b, \\ \texttt{While}[b \wedge b_1', c_1; c_2; c_4]; \\ \texttt{While}[b \wedge \neg b1', c_1; c_3; c_4]] \end{array} \qquad \{I \wedge \neg b\} \tag{8}$$

where I is a loop invariant and b_1' represents condition b_1 modified by the assignment statement c_1.
(The soundness proof of this rule is given in [23].)

What remains is to determine the relation between the polynomial invariants of the P-solvable loop (7) and the polynomial identities of the inner loops from (8). For doing so:

(i) first we show that the polynomial relations among the variables from an arbitrary iteration of the outer loop (8) are determined by the intersection of ideals of the polynomial relations of its inner loops;
(ii) next, we show that any iteration of the outer loop (8) admits the same set of polynomial relations among the loop variables;
(iii) hence for generating all polynomial invariants of (7), suffices to consider the first iteration of the outer loop (8) whose polynomials are determined by the intersection of the ideals of polynomial relations of its inner loops.

In more detail, we proceed as follows:

(i) For an arbitrary iteration of the outer loop (8), we denote by $n_1,\ n_2$ the loop counters of its first and second inner loops. The variables $x_1(n_1), \ldots, x_m(n_1)$ of the first P-solvable inner loop depend only on n_1, whereas $x_1(n_2), \ldots,\ x_m(n_2)$ of the second P-solvable inner loop only on n_2. By the algorithm from the previous section, we determine all polynomial relations for each inner loop. Since the arbitrary iteration of the outer loop is fully described by the recurrence equations of its first and second inner loops, taking the intersection of the polynomial relations of the inner loops, we obtain all polynomial relations of the arbitrary iteration of the outer loop.

(ii) The polynomial relations among the loop variables of the P-solvable inner loops do not depend on the outer loop iterations, they are obtained from the closed forms of the inner loop variables by eliminating the inner loop counters and exponential terms in the inner loop counters. For every iteration of the outer loop (8) the recurrence equations of the inner loop variables are the same, thus also the ideals of polynomial relations among the the inner loop variables remain the same for any iteration of (8). Hence, by step (i), any iteration of the outer loop (8) admits the same ideal of polynomial relations among the loop variables, and we have:

Theorem 4.1 *[23]*
The polynomial relations among the loop variables of (8) are captured by the polynomial relations among the loop variables after the first iteration of the outer loop (8).

(iii) Finally, we are able to determine (and give an algorithmic computation of) all polynomial relation of the P-solvable loop (7):

Theorem 4.2 *Polynomial Invariants of P-solvable Loops with Conditionals.*

The intersection of polynomial invariants of the P-solvable inner loops from the first iteration of (8) represents all polynomial invariants of the P-solvable loop (7).

Proof. Consider a P-solvable loop with conditional as in (7), having its recursively changed loop variables $X = \{x_1, \ldots, x_m\}$. Let us denote by n_1 and n_2 the loop counters of the P-solvable inner loops from the first iteration of (8).

- For the first inner loop, we have closed form solutions for $x_1(n_1), \ldots, x_m(n_1)$ as polynomials of n_1, exponential terms and initial values;
- Similarly, for the second inner loop, we have closed form solutions for $x_1(n_2)$, $\ldots, x_m(n_2)$ as polynomials of n_2, exponential terms and initial values that are given by the final values of the first inner loop as obtained in the previous step;
- Using the algorithm for P-solvable loops with assignments only, we determine all polynomial relations among $x_1(n_1), \ldots, x_m(n_1)$ and $x_1(n_2), \ldots, x_m(n_2)$;
- By step (i), all polynomial relations of the first iteration of (8) are determined by the intersection of the polynomials of its inner loops;
- By Theorem 4.1, the generators of the ideal of polynomials (8), thus of (7), from which any polynomial invariant can be derived, are determined by the intersection of all polynomial relations of the inner loops from the first iteration of (8).

5 Implementation of the Invariant Generation Algorithm

We have implemented in *Theorema* a procedural language, as well as a verification condition generator (`VCG`) [21,24] based on Hoare-Logic [14] and using the Weakest Precondition Strategy [6]. The constructs of the programming language are: assignments, blocks, conditionals, `For` and `While` loops, procedure calls. The user interface has simple, intuitive commands (`Program`, `Specification`, `VCG`, `Execute`). Programs are considered as procedures, without return values and with input, output and/or transient parameters. Programs are annotated with loop invariants (`Invariant`) and termination terms (`TerminationTerm`).

We illustrate our invariant generation algorithm on the following example:

Example 5.1 *Algorithm for Computing the product of two natural numbers*

```
        Specification[ "Product", Prod[↓ a, ↓ b, ↑ z],
        Pre → $(a \geq 0) \wedge (b \geq 0)$,
        Post → $(z = a * b)$]
        Program[ "Product", Prod[↓ a, ↓ b, ↑ z],
        Module[{x, y}, $x := a;\ y := b;\ z := 0;$
(A)     While[$(y \neq 0)$,
(B)        If[Odd[y] Then $z := z + x, y := y - 1$];
(C)        $x := 2 * x; y := y/2$]]]
```

Step 0: Loop Transformation (see Prop. 4.1).

```
(A)     While[$(y \neq 0)$,
(B)        While[$(y \neq 0) \wedge$ Odd[y],
               $z := z + x; y := y - 1; x := 2 * x; y := y/2$];
(C)        While[$(y \neq 0) \wedge \neg$Odd[y],
               $x := 2 * x; y := y/2$]]
```

Step 1: Solving Recurrences for the Inner Loops.

Step 1.(a): Extracting recurrence equations.

While loop (B):

$n_1 = 0, \dots, N_1$

$$\begin{cases} x(n_1+1) = 2 * x(n_1) \\ y(n_1+1) = \frac{1}{2} * y(n_1) - \frac{1}{2} \\ z(n_1+1) = z(n_1) + x(n_1) \end{cases}$$

While loop (C):

$n_2 = 0, \dots, N_2$

$$\begin{cases} x(n_2'+1) = 2 * x(n_2') \; , \\ y(n_2'+1) = \frac{1}{2} * y(n_2') \\ z(n_2'+1) = z(n_2') \end{cases}$$

where $n_2' = n_2 + N_1$ and $N_1,\ N_2$ represent the unknown bounds of the inner-loop counters $n_1,\ n_2$.

Step 1.(b): Solving recurrences.

While loop (B):

$n_1 = 0, \dots, N_1$

$$\begin{cases} x(n_1) \underset{geom.series}{=} 2^{n_1} * x(0) \\ y(n_1) \underset{C-finite}{=} \big(y(0)+1\big) * 2^{-n_1} - 1 \\ z(n_1) \underset{Gosper}{=} z(0) - x(0) + 2^{n_1} * x(0) \end{cases}$$

While loop (C):

$n_2 = 0, \dots, N_2$

$$\begin{cases} x(n_2') \underset{geom.series}{=} 2^{n_2} * x(N_1) \\ y(n_2') \underset{geom.series}{=} 2^{-n_2} * y(N_1) \\ z(n_2') = z(N_1) \end{cases}$$

Step 1.(c): Determining algebraic dependencies. Introducing new variables.

While loop (B):

$n_1 = 0, \dots, N_1$

$u(n_1) = 2^{n_1}, v(n_1) = 2^{-n_1}$

$$\begin{cases} x(n_1) = u(n_1) * x(0) \\ y(n_1) = \big(y(0)+1\big) * v(n_1) - 1 \\ z(n_1) = z(0) - x(0) + u(n_1) * x(0) \\ 0 \underset{Alg.Dep.}{=} u(n_1) * v(n_1) - 1 \end{cases}$$

While loop (C):

$n_2 = 0, \dots, N_2$

$p(n_2') = 2^{n_2}, q(n_2') = 2^{-n_2}$

$$\begin{cases} x(n_2') = p(n_2') * x(N_1) \\ y(n_2') = q(n_2') * y(N_1) \\ z(n_2') = z(N_1) \\ 0 \underset{Alg.Dep.}{=} p(n_2') * q(n_2') - 1 \end{cases}$$

Step 2: Polynomial Invariant Generation for the Outer Loop.
After eliminating the variables u, v, p, q by Gröbner basis computation, the polynomial invariant for the while loop (A) is:

$$z + x * y = a * b.$$

The automatically generated polynomial invariant properties, together with the user-asserted non-polynomial invariants (e.g. inequalities, modulo expressions, etc.) are used further by the `VCG` for verifying partial correctness of the program. The `VCG` takes the (annotated) program and its specification, and, based on the weakest precondition strategy, generates a purely logical proof obligation, i.e. a list of verification conditions, in *Theorema* syntax. For example 5.1, the `VCG` produces a universally quantified lemma with 3 proof obligations in order to prove partial correctness of the program. The verification conditions thus generated are manipulated by provers of *Theorema*, particularly, the PCS prover [2], that uses quantifier elimination, and produces human-readable proofs of the verification conditions. The proving part of the verification process is beyond the scope of the current paper.

Further Examples. We have successfully tested our method on a number of interesting number theoretic examples [23], some of them being listed in the table below. The first column of the table contains the name of the example, the second and third columns specify the applied combinatorial methods and the number of generated polynomial invariants for the corresponding example, whereas the fourth column shows the timing (in seconds) needed by the implementation on a Pentium 4, 1.6GHz processor with 512 Mb RAM.

Example	**Comb. Methods**	**Nr.Poly.**	**(sec)**
P-solvable loops with assignments only			
Division [21]	Gosper	1	0.08
Integer square root [21]	Gosper	2	0.09
Integer cubic root [29]	Gosper	2	0.15
Fibonacci [24]	Generating Functions, Alg.Dependencies	1	0.73
P-solvable loops with conditionals and assignments			
Wensley's Algorithm [31]	Gosper, geom.series, Alg.Dependcies	2	0.48
LCM-GCD computation [6]	Gosper	1	0.33
Extended GCD [29]	Gosper	3	0.65
Fermat's factorization [22]	Gosper	1	0.32
Square root [33]	C-finite, Gosper, geom.series, Alg.Dependencies	1	1.28
Binary Division [15]	C-finite, Gosper, geom.series, Alg.Dependencies	1	0.72
Floor of square root [6]	Gosper, C-finite, geom.series, Alg.Dependencies	1	1.06
Factoring Large Numbers [22]	C-finite, Gosper	1	1.9
Hardware Integer Division [30]			0.62
1st Loop	geom.series, Alg.Dependencies	3	
2nd Loop	Gosper, geom. series, Alg.Dependencies	2	

6 Conclusions

A framework combining combinatorics, algebraic relations and logic is presented for generating loop invariants for a family of imperative programs operating on numbers. The framework is implemented in the *Theorema* system. It is indeed possible to generate polynomial equations as loop invariants, which can be subsequently used for verifying properties of programs. A collection of examples successfully worked out using the framework is presented in [23]. The imperative verification environment implemented in the *Theorema* system demonstrates the applicability and the usefulness of several algebraic and combinatorial techniques for the automated generation of polynomial invariants, and, in more general terms, the advantages of combining techniques from computational logic with techniques from computer algebra.

So far, the focus has been on generating polynomial equations as loop invariants. We believe that it should be possible to identify and generate polynomial inequalities in addition to polynomial equations, as invariants as well. We have been investigating the manipulation of pre- and postconditions, and other annotations of programs, if available, along with conditions in loops and conditional statements, as well as the simple fact that no loop is executed less than 0 times. Quantifier elimination methods on theories, including the theory of real closed fields, should be helpful. Of course, we are also interested in generalizing the framework to programs on nonnumeric data structures.

Acknowledgement. The authors wish to thank Manuel Kauers (RISC-Linz Combinatorics group) for his help and comments.

References

[1] B. Buchberger. Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985.

[2] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, N. Popov K.Nakagawa, J. Robu, W. Windsteiger, F. Piroi, and M. Rosenkranz. Theorema: Towards Systematic Mathematical Theory Exploration. *J. of Applied Logic, Special Issue on Mathematical Assistant Systems*, 2005. To appear.

[3] G. E. Collins. Quantifier Elimination for the Elementary Theory of Real Closed Fields by Cylindrical Algebraic Decomposition. *LNCS*, 33:134–183, 1975.

[4] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[5] D. Cox, J. Little, and D. O'Shea. *Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer, 2nd edition, 1998.

[6] E. W. Dijkstra. *A Discipline of Programming.* Prentince-Hall, 1976.

[7] B. Elspas, M. W. Green, K. N. Lewitt, and R. J. Waldinger. Research in Interactive Program - Proving Techniques. Technical report, Stanford Research Institute, 1972.

[8] H. Enderton. *Mathematical Logic, an Introduction.* Academic Press, 1992.

[9] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs.* American Mathematical Society, 2003.

[10] R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.

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

[12] R. W. Gosper. Decision Procedures for Indefinite Hypergeometric Summation. *Journal of Symbolic Computation*, 75:40–42, 1978.

[13] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley Publishing Company, 2nd edition, 1989. pg. 306–330.

[14] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[15] A. Kaldewaij. *Programming. The Derivation of Algorithms.* Prentince-Hall, 1990.

[16] D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. In *Proc. of ACA*, 2004.

[17] M. Karr. Affine Relationships Amomg Variables of Programs. *Acta Informatica*, 6:133–151, 1976.

[18] S. Katz and Z. Manna. Logical Analysis of Programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[19] M. Kauers. SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *Journal of Symbolic Computation*, 41:1039–1057, 2006.

[20] M. Kauers and B. Zimmermann. Computing the Algebraic Relations of C-finite Sequences and Multisequences. Technical Report 2006-24, SFB F013, Linz, Austria, 2006. (submitted).

[21] M. Kirchner. Program Verification with the Mathematical Software System *Theorema.* Technical Report 99–16, RISC-Linz, Austria, 1999.

[22] D. E. Knuth. *The Art of Computer Programming*, volume 2. *Seminumerical Algorithms.* Addison-Wesley, 3rd edition, 1998.

[23] L. Kovacs. Finding Polynomial Invariants for Imperative Loops in the Theorema System. Technical Report 06-03, RISC-Linz, Austria, 2006.

[24] L. Kovacs and T. Jebelean. Finding Polynomial Invariants for Imperative Loops in the Theorema System. In *Proc. of Verify'06, FLoC'06*, 2006.

[25] L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Proc. of ISOLA 2006*, 2006. To appear.

[26] M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*, 2006.

[27] M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *SAS 2002*, volume 2477 of *LNCS*, pages 4–19, 2002.

[28] P. Paule and M. Schorn. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *J. Symbolic Computation*, 20(5–6):673–698, 1995.

[29] E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC'04*, 2004.

[30] S. Sankaranaryanan, B. S. Henry, and Z. Manna. Nonlinear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL'04*, 2004.

[31] B. Wegbreit. The Synthesis of Loop Predicates. *Communication of the ACM*, 2(17):102–112, 1974.

[32] D. Zeilberger. A Holonomic Systems Approach To Special Functions. *Journal of Computational and Applied Mathematics*, 32:321–368, 1990.

[33] K. Zuse. *The Computer - My Life*. Springer, 1993.