Reasoning Algebraically About P-Solvable Loops

Laura Kovács*

EPFL, Switzerland
laura.kovacs@epfl.ch

Abstract. We present a method for generating polynomial invariants for a subfamily of imperative loops operating on numbers, called the P-solvable loops. The method uses algorithmic combinatorics and algebraic techniques. The approach is shown to be complete for some special cases. By completeness we mean that it generates a set of polynomial invariants from which, under additional assumptions, any polynomial invariant can be derived. These techniques are implemented in a new software package `Aligator` written in *Mathematica* and successfully tried on many programs implementing interesting algorithms working on numbers.

1 Introduction

This paper discusses an approach for automatically generating polynomial equations as loop invariants by combining advanced techniques from algorithmic combinatorics and polynomial algebra. Polynomial invariants 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 problems [21], e. g., constant propagation, discovery of symbolic constants, discovery of loop induction variables, etc.

Exploiting the symbolic manipulation capabilities of the computer algebra system *Mathematica*, the approach is implemented in a new software package called `Aligator` [17]. By using several combinatorial packages developed at RISC, `Aligator` includes algorithms for solving special classes of recurrence relations (those that are either Gosper-summable or C-finite) and generating polynomial dependencies among algebraic exponential sequences. Using `Aligator`, a complete set of polynomial invariants is successfully generated for numerous imperative programs working on numbers [17]; some of these examples are presented in this paper.

The key steps of our method for invariant generation are as follows.

(i) Assignment statements from the loop body are extracted. They form a system of *recurrence equations* describing the behavior of those loop variables that are changing at each loop iteration.

* The results presented here were obtained at the Research Institute for Symbolic Computation (RISC), Linz, Austria. The work was supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project, and by FWF (Austrian National Science Foundation) - SFB project F1302.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 249–264, 2008.

(ii) Methods of algorithmic combinatorics are used to *solve exactly* the recurrence equations, yielding the *closed form* for each loop variable.
(iii) *Algebraic dependencies* among possible exponential sequences of algebraic numbers occurring in the closed forms of the loop variables are derived using algebraic and combinatorial methods.
As a result of these steps, every program variable can be expressed as a polynomial of the initial values of variables (those when the loop is entered), the loop counter, and some new variables, where there are algebraic dependencies among the new variables.
(iv) Loop counters are then eliminated by polynomial methods to derive a finite set of polynomial identities among the program variables as invariants. From this finite set, under additional assumptions when the loop body contains conditionals branches, any polynomial identity that is a loop invariant can be derived.

In our approach to invariant generation, a family of imperative loops, called *P-solvable* (to stand for polynomial-solvable), is identified, for which test conditions in the loop and conditional branches are ignored and the value of each program variable is expressed as a polynomial of the initial values of variables, loop counter, and some new variables where there are algebraic dependencies among the new variables. We show that for such loops, polynomial invariants can be automatically generated. Many non-trivial algorithms working on numbers can be naturally implemented using P-solvable loops.

Further, if the bodies of these loops consist only of assignments whose right hand sides are polynomials of certain shape, then the approach generates a *complete* set of polynomial invariants of the loop from which any other polynomial invariant can be obtained.

Moreover, if the P-solvable loop bodies contain conditional branches as well, under additional assumptions the approach is proved to be also *complete* in generating a set of polynomial invariants of the loop from which any further polynomial invariant can be derived. We could not find any example of a P-solvable loop with conditional branches and assignments for which our approach fails to be complete. We thus conjecture that the imposed constraints cover a large class of imperative programs, and the completeness proof of our approach without the additional assumptions is a challenging task for further research.

The automatically obtained invariant assertions, together with the user-asserted non-polynomial invariant properties, can be subsequently used for proving the partial correctness of programs by generating appropriate verification conditions as first-order logical formulas. This verification process is supported in an imperative verification environment implemented in the *Theorema* system [2].

This paper extends our earlier experimental papers [18, 19] by the completeness and correctness results of the invariant generation algorithm, and by a complete treatment of the affine loops. We omit proofs, they can be found in [17].

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 concrete examples. Section 5 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 [8,12]. 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 [22, 25, 26, 11] have proposed methods for automatically computing polynomial invariant identities using polynomial ideal theoretic algorithms.

In [22,26], the invariant generation problem is translated to a constraint solving problem. In [26], non-linear (algebraic) invariants are proposed as templates with parameters; constraints on parameters are generated (by forward propagation) and solved using the theory of ideals over polynomial rings. In [22], 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. This is also the case in [11] where a method for invariant generation using *quantifier-elimination* [3] 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.

A related approach for polynomial invariant generation without any a priori bound on the degree of polynomials is presented in [25]. 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 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, unlike [25], where polynomial dependencies could be derived only for a special case of algebraic exponentials, namely, for rationals. Finally, for eliminating the loop counter and the variables standing for the exponential sequences in the loop counter from these closed form solutions expressed as polynomials, a Gröbner basis computation is performed; however, we do not need to perform Gröbner basis computations iteratively. Contrarily to [25] where completeness is always guaranteed, the completeness of our method for loops with conditional branches and assignments

is proved only under additional assumptions over ideals of polynomial invariants. It is worth to be mentioned though that these additional constraints cover a wide class of loops, and we could not find any example for which the completeness of our approach is violated.

3 Theoretical Preliminaries

This section contains some definitions and facts about linear recurrences, ideals and algebraic dependencies. For additional details see [5, 7].

In what follows, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ denote respectively the set of natural, integer, rational and real numbers. Throughout this paper we assume that $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{Q}$, $\mathbb{R}$, etc.) and denote by $\bar{\mathbb{K}}$ its algebraic closure. All rings are commutative.

Definition 3.1. Sequences and Recurrences

- A univariate *sequence* in $\mathbb{K}$ is a function $f : \mathbb{N} \to \mathbb{K}$. By $f(n)$ we denote both the value of f at the point n and the whole sequence f itself.
- A *recurrence* for a sequence $f(n)$ is

$$f(n+r) = R(f(n), f(n+1), \ldots, f(n+r-1), n) \quad (n \in \mathbb{N}),$$

for some function $R : \mathbb{K}^{r+1} \to \mathbb{K}$, where r is a natural number, called the *order* of the recurrence.

The recurrence equation of $f(n)$ allows the computation of $f(n)$ for any $n \in \mathbb{N}$: first the previous values $f(1), \ldots, f(n-1)$ are determined, and then $f(n)$ is obtained. A *solution to the recurrence* would be thus more suitable for getting the value of $f(n)$ for any n as a function of the recurrence index n. That is a *closed form* solution of $f(n)$. Since finding closed form expressions of recurrences in the general case is undecidable, it is necessary to distinguish among the type of recurrence equations. In what follows, several classes of recurrences will be briefly presented together with the algorithmic methods for solving them.

Definition 3.2. Gosper-summable and C-finite Recurrences [9, 28]

1. A *Gosper-summable recurrence* f(n) in $\mathbb{K}$ is a recurrence

$$f(n+1) = f(n) + h(n+1) \quad (n \in \mathbb{N}), \tag{1}$$

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

2. A *C-finite recurrence* f(n) in $\mathbb{K}$ is a (homogeneous) linear recurrence with constant coefficients

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

where $r \in \mathbb{N}$ 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

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

Computation of Closed Forms
(i) The closed-form solution of a Gosper-summable recurrence can be exactly computed [9]; for doing so, we use the recurrence solving package `zb` [23], implemented in *Mathematica* by the RISC Combinatorics group. For example, given the Gosper-summable recurrence $f(n+1) = f(n) + \frac{1}{2^{n+1}},\ n \geq 0$, with the initial value $f(0)$, we obtain its closed form $f(n) = f(0) + 2 - 2 * 2^{-n}$.
(ii) A crucial and elementary fact about a C-finite recurrence $f(n)$ in $\mathbb{K}$ is that it always admits a closed form solution [7]. Its closed form is

$$f(n) = p_1(n)\theta_1^n + \cdots + p_s(n)\theta_s^n, \tag{3}$$

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 $\theta_i,\ i = 1, \ldots, s$. The closed form (3) of $f(n)$ is called a *C-finite expression*.

An additional nice property of C-finite recurrences is that an *inhomogeneous* linear recurrence with constant coefficients

$$f(n+r) = a_0 f(n) + a_1 f(n+1) + \cdots + a_{r-1} f(n+r-1) + g(n) \quad (n \in \mathbb{N}),$$

where $a_0, \ldots, a_{r-1} \in \mathbb{K}$ and $g(n) \neq 0$ is a C-finite expression in n, can always be transformed into an equivalent (homogenous) C-finite recurrence. Hence, its closed form can always be computed.

For obtaining the closed form solutions of (C-finite) linear recurrences we use the `SumCracker` package [13], a *Mathematica* implementation by the RISC Combinatorics group. For example, given the linear recurrence $f(n+1) = \frac{1}{2} * f(n) + 1,\ n \geq 0$, with initial value $f(0)$, we obtain its closed form $f(n) = \frac{1}{2^n} * f(0) - \frac{2}{2^n} + 2$.

In this paper we consider the ring $\mathbb{K}[x_1, \ldots, x_m]$ of polynomials in the loop variables $x_1, \ldots, x_m$ with coefficients in $\mathbb{K}$, and perform operations over ideals of $\mathbb{K}[x_1, \ldots, x_m]$. Thus, it is necessary for our approach to 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 with special properties making 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 3.3. Algebraic Dependencies among Exponential Sequences [14]
Let $\theta_1, \ldots, \theta_s \in \bar{\mathbb{K}}$ be algebraic numbers, and their corresponding exponential sequences $\theta_1^n, \ldots, \theta_s^n,\ n \in \mathbb{N}$.

An *algebraic dependency* (or *algebraic relation*) of these sequences over $\bar{\mathbb{K}}$ is a polynomial $p \in \bar{\mathbb{K}}[x_1, \ldots, x_s]$ in s distinct variables $x_1, \ldots, x_s$, i.e. in as many distinct variables as exponential sequences, such that p vanishes when variables are substituted by the exponential sequences, namely:

$$p(\theta_1^n, \ldots, \theta_s^n) = 0, \quad \forall n \in \mathbb{N}. \tag{4}$$

Note that the multiplicative relations among $\theta_1, \ldots, \theta_s$ imply corresponding relations among $\theta_1^n, \ldots, \theta_s^n$. Further, by results of [14], the ideal $I(\theta_1^n, \ldots, \theta_s^n)$ of algebraic dependencies among the sequences $\theta_1^n, \ldots, \theta_s^n$ is the same as the ideal $I(n, \theta_1^n, \ldots, \theta_s^n)$.

For automatically determining the *ideal* $I(n, \theta_1^n, \ldots, \theta_s^n)$ *of algebraic dependencies* among $\theta_1^n, \ldots, \theta_s^n$ we use the `Dependencies` package [14] implemented in *Mathematica* by the RISC combinatorics group. For example, $\theta_1^{2n} - \theta_2^n = 0$ is an algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 4$ and there is no algebraic dependency among the exponential sequences of $\theta_1 = 2$ and $\theta_2 = 3$.

4 Generation of Invariant Polynomial Identities

As observed already by [25], the set of polynomial invariants forms a polynomial ideal. The challenging task is thus to determine the *polynomial invariant ideal*.

The algorithm for polynomial invariant generation presented in this paper combines computer algebra and algorithmic combinatorics in such a way that at the end of the invariant generation process valid polynomial assertions of a *P-solvable loop* are automatically obtained. Moreover, under additional assumptions for loops with conditional branches, our approach is proved to be complete: it returns a *basis* for the polynomial invariant ideal.

In our approach for generating polynomial invariants, test conditions in the loops and conditionals are ignored. This turns the considered loops into *non-deterministic* program fragment.

For any conditional statement `If`$[b$ `Then` S_1 `Else` $S_2]$, where S_1 and S_2 are sequences of assignments, we will omit the boolean condition b and write it in the form `If`$[\ldots$ `Then` S_1 `Else` $S_2]$ to mean the non-deterministic program $S_1|S_2$. Likewise, we omit the condition b from a loop `While`$[b, S]$, where S is a sequence of assignments, and will write it in the form

$$\texttt{While}[\ldots, S] \tag{5}$$

to mean the non-deterministic program S^*. A detailed presentation of the syntax and semantics of considered non-deterministic programs can be found in [17].

Ignoring the tests in the conditional branches means that either branch is executed in every possible way, whereas ignoring the test condition of the loop means the loop is executed arbitrarily many nonzero times. We will refer to the loop obtained in this way by dropping the loop condition and all test conditions also as a *P-solvable loop.* In the rest of this paper we will focus on *non-deterministic P-solvable loops with assignments and conditional branches with ignored conditions*, written as below.

$$\texttt{While}[\ldots, \texttt{If}[\ldots \texttt{ Then } S_1];\ \ldots\ ; \texttt{If}[\ldots \texttt{ Then } S_k]]. \tag{6}$$

The definition of P-solvable loops is available in our earlier conference papers [18, 19]. Informally, an imperative loop is *P-solvable* if the closed form solution of the loop variables are polynomials of the initial values of variables, the loop counter, and some new variables, where there are algebraic dependencies among the new variables. The class of P-solvable loops includes the simple situations when the expressions in the assignment statements are affine mappings, as stated below.

Theorem 4.1. Affine loops are P-solvable

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

P-solvable Loops with Assignments Only. We denote by $n \in \mathbb{N}$ the loop counter, by $X = \{x_1, \ldots, x_m\}$ $(m > 1)$ the recursively changed loop variables whose initial values (before entering the loop) are denoted by X_0. Our method for automatically deriving a *basis of the polynomial invariant ideal for P-solvable loops with assignments only* is presented in Algorithm 4.1.

Algorithm 4.1 "receives" as input a P-solvable loop with assignments only ($k = 1$ in (6)), and starts first with extracting and solving the recurrence equations of the loop variables. The closed forms of the variables are thus determined (steps 1-3 of Algorithm 4.1). Next, it computes the set A of generators for the ideal of algebraic dependencies among the exponential sequences from the closed form system (step 4 of Algorithm 4.1). Finally, from the ideal I generated by the polynomial system of closed forms and A, the ideal G of *all polynomial relations* among the loop variables is computed by elimination using Gröbner basis w.r.t. a suitable elimination order (steps 5-7 of Algorithm 4.1).

Algorithm 4.1 P-solvable Loops with Assignments Only
Input: Imperative P-solvable loop (5) with only assignment statements S, having its recursively changed variables $X = \{x_1, \ldots, x_m\}$ with initial values X_0
Output: The ideal $G \trianglelefteq \mathbb{K}[X]$ of polynomial invariants among X
Assumption: The recurrence equations of X are of order at least 1, $n \in \mathbb{N}$

1 Extract the recurrence equations of the loop variables

I. Recurrence Solving.

2 Identify the type of recurrences and solve them by the methods from page 253
3 Using the P-solvable loop property, the closed form system is

$$\begin{cases} x_1[n] = \ q_1(n, \theta_1^n, \ldots, \theta_s^n) \\ \vdots \\ x_m[n] = q_m(n, \theta_1^n, \ldots, \theta_s^n) \end{cases}, \text{ where } \begin{array}{l} \theta_j \in \bar{\mathbb{K}},\ q_i \in \bar{\mathbb{K}}[n, \theta_1^n, \ldots, \theta_s^n], \\ q_i \text{ are parameterized by } X_0, \\ j = 1, \ldots, s,\ \ i = 1, \ldots, m \end{array}$$

4 Compute a basis A for the ideal of algebraic dependencies among $n, \theta_1^n, \ldots, \theta_s^n$.

$$\text{Conform page 253, } \ \langle A \rangle = I(n, \theta_1^n, \ldots, \theta_s^n)$$

5 Denote $z_0 = n, z_1 = \theta_1^n, \ldots, z_s = \theta_s^n$. Thus $\langle A \rangle \trianglelefteq \bar{\mathbb{K}}[z_0, \ldots, z_s]$ and

$$\begin{cases} x_1 = \ q_1(z_0, z_1, \ldots, z_s) \\ \vdots \\ x_m = q_m(z_0, z_1, \ldots, z_s) \end{cases}, \text{ where } \begin{array}{l} q_i \in \bar{\mathbb{K}}[z_0, z_1, \ldots, z_s],\ i = 1, \ldots, m, \\ q_i \text{ are parameterized by } X_0. \end{array}$$

II. Polynomial Invariant Generation.

6 Consider $I = \langle x_1 - q_1(z_0, \ldots, z_s), \ldots, x_m - q_m(z_0, \ldots, z_s)\rangle + \langle A\rangle$.

$$\text{Thus } I \subset \bar{\mathbb{K}}[z_0, z_1, \ldots, z_s, x_1, \ldots, x_m]$$

7 **return** $G = I \cap \mathbb{K}[x_1, \ldots, x_m]$.

Theorem 4.2. Algorithm 4.1 is correct. Its output G satisfies

1. $G \trianglelefteq \mathbb{K}[x_1, \ldots, x_m]$;
2. every polynomial relation from G is a polynomial invariant among the P-solvable loop variables $x_1, \ldots, x_m$ over $\mathbb{K}[X]$;
3. any polynomial invariant among the P-solvable loop variables $x_1, \ldots, x_m$ over $\mathbb{K}[X]$ can be derived from (the generators of) G.

The restrictions at the various steps of Algorithm 4.1 are crucial. If 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. Thus, Algorithm 4.1 can be applied only to P-solvable loops whose assignment statements describe either Gosper-summable or C-finite recurrences.

Example 4.3. Given the loop

$$\texttt{While}[\ldots,\ a := a + b; y := y + d/2; b := b/2; d := d/2],$$

its polynomial invariants, by applying Algorithm 4.1 and using `Aligator`, are obtained as follows.

Step 1:
$$\begin{cases} a[n+1] = a[n] + b[n] \\ y[n+1] = y[n] + d[n]/2 \\ b[n+1] = b[n]/2 \\ d[n+1] = d[n]/2 \end{cases}$$

Steps 2,3:
$$\begin{cases} a[n] \underset{Gosper}{=} a[0] + 2 * b[0] - 2 * b[0] * 2^{-n} \\ b[n] \underset{C-finite}{=} b[0] * 2^{-n} \\ d[n] \underset{C-finite}{=} d[0] * 2^{-n} \\ y[n] \underset{Gosper}{=} y[0] + d[0] - d[0] * 2^{-n} \end{cases}$$

where $a[0], b[0], d[0], y[0]$ denote the initial values of a, b, d, y before the loop.
Steps 4, 5: $z_1 = 2^{-n}, z_2 = 2^{-n}, z_3 = 2^{-n}, z_4 = 2^{-n}$

$$\begin{cases} a = a[0] + 2 * b[0] - 2 * b[0] * z_1 \\ b = b[0] * z_2 \\ d = d[0] * z_3 \\ y = y[0] + d[0] - d[0] * z_4 \end{cases} \quad \text{with} \quad \text{algebraic dependencies: } \begin{cases} z_1 - z_4 = 0 \\ z_2 - z_4 = 0 \\ z_3 - z_4 = 0 \end{cases}$$

Steps 6, 7: The Gröbner basis computation with $z_1 \succ z_2 \succ z_3 \succ z_4 \succ a \succ b \succ d \succ y$ yields:
$G = \langle d + y - d[0] - y[0], y\,b[0] + b\,d[0] - b[0]d[0] - b[0]y[0], a + 2b - a[0] - 2b[0]\rangle$.

Based on Theorems 4.1 and 4.2, we finally state the theorem below.

Theorem 4.4. The ideal of polynomial invariants for an affine loop is algorithmically computable by Algorithm 4.1.

P-solvable Loops with Conditionals and Assignments. We consider a generalization of Algorithm 4.1. for P-solvable loops with conditional branches and assignments.

The starting point of our approach is to do first program transformations (see Theorem 4.5). Namely, transform the P-solvable loop with conditional branches, i.e. outer loop, into nested P-solvable loops with assignments only, i.e. inner loops. Further, we apply steps of Algorithm 4.1 to reason about the inner loops such that at the end we derive polynomial invariants of the outer loop. Moreover, under the additional assumptions introduced in Theorem 4.12, we prove that our approach is complete. Namely, it returns a basis for the polynomial invariant ideal for some special cases of P-solvable loops with conditional branches and assignments. It is worth to be mentioned that the imposed assumptions cover a wide class of imperative programs (see [17] for concrete examples). Moreover, we could not yet find any example of a P-solvable loop for which the completeness of our approach is violated.

Theorem 4.5. Let us consider the following two loops:

$$\texttt{While}[b, s_0; \texttt{If}[b_1\ \texttt{Then}\ s_1\ \texttt{Else}\ \ldots\ \texttt{If}[b_{k-1}\ \texttt{Then}\ s_{k-1}\ \texttt{Else}\ s_k]\ldots]; s_{k+1}] \quad (7)$$

and

$$\begin{array}{l} \texttt{While}[b, \\ \texttt{While}[b \wedge b_1',\ s_0; s_1; s_{k+1}]; \\ \ldots \\ \texttt{While}[b \wedge \neg b_1' \wedge \cdots \wedge \neg b_{k-1}',\ s_0; s_k; s_{k+1}]], \end{array} \quad (8)$$

where $s_0,\ s_1,\ \ldots,\ s_k,\ s_{k+1}$ are sequences of assignments, and $b_i' = \mathrm{wp}(s_0, b_i)$ is the weakest precondition of s_0 with postcondition b_i, $i = 1, \ldots, k-1$.

Then any formula I is an invariant of the first loop if and only if it is an invariant of the second loop and all of its inner loops.

Since in our approach for invariant generation tests are ignored in the loop and conditional branches, the loop (7) can be equivalently written as (6), by denoting $S_i = s_0; s_i; s_{k+1}$. Further, using our notation for basic non-deterministic programs mentioned on page 254, the outer loop (8) can be written as $(S_1|S_2|\ldots|S_k)^*$. Based on Theorem 4.5, an imperative loop having $k \geq 1$ *conditional branches and assignment statements only* is called *P-solvable* if the inner loops obtained after performing the transformation rule from Theorem 4.5 are P-solvable.

Example 4.6. Consider the loop implementing Wensley's algorithm for real division [27].

$$\begin{array}{l} \texttt{While}[(d \geq Tol), \\ \quad \texttt{If}[(P < a + b) \\ \quad\quad \texttt{Then}\ b := b/2; d := d/2 \\ \quad\quad \texttt{Else}\ a := a + b; y := y + d/2; b := b/2; d := d/2]]. \end{array} \quad (9)$$

After applying Theorem 4.5 and omitting all test conditions, the obtained nested loop system is as follows.

$$\begin{array}{ll} & \texttt{While}[\ldots, \\ S_1: & \texttt{While}[\ldots,\ b := b/2; d := d/2]; \\ S_2: & \texttt{While}[\ldots,\ a := a + b; y := y + d/2; b := b/2; d := d/2]]. \end{array}$$

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, the main steps of our algorithm are as follows.

(i) Firstly, we determine the ideal of polynomial relations for an *arbitrary iteration* of the outer loop (8) (see Theorem 4.8).
(ii) Finally, from the ideal of polynomial relations after the *first* iteration of the outer loop (8), we keep only the *polynomial invariants* for the P-solvable loop (7) (see Theorem 4.10).

Moreover, under the *additional assumptions* of Theorem 4.12, the polynomial invariants thus obtained form a *basis for the polynomial invariant ideal* of the P-solvable loop (7).

In more detail, we proceed as follows. (i) In the general case of a P-solvable loop (7) with a nested conditional statement having $k \geq 1$ conditional branches, by applying Theorem 4.5, we obtain an outer loop (8) with k P-solvable inner loops $S_1, \ldots, S_k$. Thus an *arbitrary iteration* of the outer loop is described by an arbitrary sequence of the k P-solvable loops. Since the tests are ignored, for any iteration of the outer loop we have $k!$ possible sequences of inner P-solvable loops.

Let us denote the set of permutations of length k over $\{1, \ldots, k\}$ by $\mathfrak{S}_k$. Consider a permutation $W = (w_1, \ldots, w_k) \in \mathfrak{S}_k$ and a sequence of numbers $J = \{j_1, \ldots, j_k\} \in \mathbb{N}^k$. Then we write $S_W^J = S_{w_1}^{j_1}; S_{w_2}^{j_2}; \ldots; S_{w_k}^{j_k}$ to denote an *arbitrary iteration of the outer loop*, i.e. an arbitrary sequence of the k inner loops. By S_i^j we mean the sequence of assignments $\underbrace{S_i; \ldots; S_i}_{j \text{ times}}$.

Using steps 1-4 of Algorithm 4.1, for each P-solvable inner loop $S_{w_i}^{j_i}$ from S_W^J we obtain their system of closed forms together with their ideal of algebraic dependencies among the exponential sequences (steps 1-4 of Algorithm 4.2). Further, the system of closed forms of loop variables after S_W^J is obtained by *merging* the closed forms of its inner loops. Merging is based on the fact that the initial values of the loop variables corresponding to the inner loop $S_{w_{i+1}}^{j_{i+1}}$ are given by the final values of the loop variables after $S_{w_i}^{j_i}$ (step 5 of Algorithm 4.2). In [17] we showed that merging of closed forms of P-solvable inner loops yields a polynomial closed form system as well.

We can now compute the ideal of valid polynomial relations among the loop variables X with initial values X_0 corresponding to the sequence of assignments $\underbrace{S_{w_1}; \ldots; S_{w_1}}_{j_1 \text{ times}}; \underbrace{S_{w_2}; \ldots; S_{w_2}}_{j_2 \text{ times}}; \ldots \ \ldots; \underbrace{S_{w_k}; \ldots; S_{w_k}}_{j_k \text{ times}}$. Using notation introduced on page 254, we thus compute the ideal of valid polynomial relations after $S_{w_1}^*; \ldots; S_{w_k}^*$. This is presented in Algorithm 4.2.

Algorithm 4.2 Polynomial Relations of a P-solvable Loop Sequence
Input: k P-solvable inner loops $S_{w_1}, \ldots, S_{w_k}$
Output: The ideal $G \trianglelefteq \mathbb{K}[X]$ of polynomial relations among X with initial values X_0 after $S_{w_1}^*; \ldots; S_{w_k}^*$
Assumption: S_{w_i} are sequences of assignments, $w_i \in \{1, \ldots, k\}, j_i \in \mathbb{N}, k \geq 1$

1 **for** each $S_{w_i}^{j_i},\ i = 1, \ldots, k$ **do**
2 Apply steps 1-3 of Algorithm 4.1 for determining the closed form of $S_{w_i}^{j_i}$
3 Compute the ideal A_{w_i} of algebraic dependencies for $S_{w_i}^{j_i}$
4 **endfor**
5 Compute the merged closed form of $S_{w_1}^{j_1}; \ldots; S_{w_k}^{j_k}$:

$$\begin{cases} x_1[j_1, \ldots, j_k] = f_1(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_k, \theta_{w_k 1}^{j_k}, \ldots, \theta_{w_k s}^{j_k}) \\ \vdots \\ x_m[j_1, \ldots, j_k] = f_m(j_1, \theta_{w_1 1}^{j_1}, \ldots, \theta_{w_1 s}^{j_1}, \ldots \ldots, j_k, \theta_{w_k 1}^{j_k}, \ldots, \theta_{w_k s}^{j_k}) \end{cases}, \text{where}$$

$f_l \in \bar{\mathbb{K}}[z_{10}, \ldots, z_{1s}, \ldots \ldots, z_{k0}, \ldots, z_{ks}]$,
the variables $z_{i0}, \ldots, z_{is}$ are standing for the C-finite sequences $j_i, \theta_{w_i 1}^{j_i}, \ldots, \theta_{w_i s}^{j_i}$,
the coefficients of f_l are given by the initial values before $S_{w_1}^{j_1}; \ldots; S_{w_k}^{j_k}$

6 $A_* = \sum_{i=1}^{k} A_{w_i}$
7 $I = \langle x_1 - f_1, \ldots, x_m - f_m \rangle + A_* \subset \bar{\mathbb{K}}[z_{10}, \ldots \ldots, z_{ks}, x_1, \ldots, x_m]$
8 **return** $G = I \cap \mathbb{K}[x_1, \ldots, x_m]$.

Elimination of $z_{10}, \ldots, z_{ks}$ at step 8 is performed by Gröbner basis computation of I w.r.t. an elimination order $\succ$ such that $z_{10} \succ \cdots \cdots \succ z_{ks} \succ x_1 \cdots \succ x_m$.

Example 4.7. For Example 4.6, the steps of Algorithm 4.2 are presented below.

Steps 1-4. Similarly to Example 4.3, the closed form systems of the inner loops S_1 and S_2 are as follows.

Inner loop S_1:
$j_1 \in \mathbb{N}$
$z_{11} = 2^{-j_1}, z_{12} = 2^{-j_1}$

$$\begin{cases} a[j_1] = a[0_1] \\ b[j_1] \underset{C-finite}{=} b[0_1] * z_{11} \\ d[j_1] \underset{C-finite}{=} d[0_1] * z_{12} \\ y[j_1] = y[0_1] \end{cases}$$

Inner loop S_2:
$j_2 \in \mathbb{N}$
$z_{21} = 2^{-j_2}, z_{22} = 2^{-j_2}, z_{23} = 2^{-j_2}, z_{24} = 2^{-j_2}$

$$\begin{cases} a[j_2] \underset{Gosper}{=} a[0_2] + 2 * b[0_2] - 2 * b[0_2] * z_{21} \\ b[j_2] \underset{C-finite}{=} b[0_2] * z_{22} \\ d[j_2] \underset{C-finite}{=} d[0_2] * z_{23} \\ y[j_2] \underset{Gosper}{=} y[0_2] + d[0_2] - d[0_2] * z_{24}, \end{cases}$$

with the computed algebraic dependencies

$$\{ z_{11} - z_{12} = 0 \qquad \text{and} \qquad \begin{cases} z_{21} - z_{24} = 0 \\ z_{22} - z_{24} = 0 \\ z_{23} - z_{24} = 0, \end{cases}$$

where $X_{01} = \{a[0_1], b[0_1], d[0_1], y[0_1]\}$ and $X_{02} = \{a[0_2], b[0_2], d[0_2], y[0_2]\}$ are respectively the initial values of a, b, d, y before entering the inner loops S_1 and S_2.

Steps 5-6. For the inner loop sequence $S_1^{j_1}; S_2^{j}$ the initial values X_{02} are given by the values $a[j_1], b[j_1], d[j_1], y[j_1]$ after $S_1^{j_1}$. Hence, the merged closed form of $S_1^{j_1}; S_2^{j_2}$ is

given below. For simplicity, let us rename the initial values X_{01} to respectively $X_0 = \{a[0], b[0], d[0], y[0]\}$.

$$\begin{cases} a[j_1, j_2] = a[0] + 2 * b[0] * z_{11} - 2b[0] * z_{21} * z_{11} \\ b[j_1, j_2] = b[0] * z_{22} * z_{11} \\ d[j_1, j_2] = d[0] * z_{12} * z_{23} \\ y[j_1, j_2] = y[0] + d[0] * z_{12} - d[0] * z_{24} * z_{12}, \end{cases} \tag{10}$$

with the already computed algebraic dependencies

$$A_* = \langle z_{11} - z_{12}, z_{21} - z_{24}, z_{22} - z_{24}, z_{23} - z_{24} \rangle. \tag{11}$$

Steps 7, 8. From (10) and (11), by eliminating $z_{11}, z_{12}, z_{21}, z_{22}, z_{23}, z_{24}$, we obtain the ideal of polynomial relations for $S_1^{j_1}; S_2^{j_2}$, as below.

$$G = \langle\, -b[0] * d + b * d[0], a * d[0] - a[0] * d[0] - 2 * b[0] * y + 2 * b[0] * y[0], \\ a * d - a[0] * d - 2 * b * y + 2 * b * y[0] \rangle.$$

In order to get all polynomial relations among the loop variables X with initial values X_0 corresponding to an arbitrary iteration of the outer loop (8), we need to apply Algorithm 4.2 on each possible sequence of k inner loops that are in a number of $k!$. This way, for each sequence of k inner loops we get the ideal of their polynomial relations among the loop variables X with initial values X_0 (step 3 of Algorithm 4.3). Using ideal theoretic results, by taking the *intersection* of all these ideals, we derive *the ideal of polynomial relations among the loop variables X with initial values X_0 that are valid after any sequence of k P-solvable inner loops* (step 4 of Algorithm 4.3). The intersection ideal thus obtained is the ideal of polynomial relations among the loop variables X with initial values X_0 *after an arbitrary iteration of the outer loop* (8). This can be algorithmically computed as follows.

Algorithm 4.3 Polynomial Relations for an Iteration of (8)
Input: P-solvable loop (8) with P-solvable inner loops $S_1, \ldots, S_k$
Output: The ideal $PI \subset \mathbb{K}[X]$ of the polynomial relations among X with initial values X_0 corresponding to an arbitrary iteration of (8)
Assumption: X_0 are the initial values of X before the arbitrary iteration of (8)

1 $PI =$ Algorithm 4.2$\big(S_1, \ldots, S_k\big)$
2 **for** each $W \in \mathfrak{S}_k \setminus \{(1, \ldots, k)\}$ **do**
3 $\quad G =$ Algorithm 4.2$\big(S_{w_1}, \ldots, S_{w_k}\big)$
4 $\quad PI = PI \cap G$
5 **endfor**
6 **return** PI.

Theorem 4.8. Algorithm 4.3 is correct. It returns the generators for the ideal PI of polynomial relations among the loop variables X with initial values X_0 after a possible iteration of the outer loop (8).

Example 4.9. Similarly to Example 4.7, we compute the ideal of polynomial relations for $S_2^{j_2}; S_1^{j_1}$ for Example 4.6. Further, we take the intersection of the ideals of polynomial relations for $S_1^{j_1}; S_2^{j_2}$ and $S_2^{j_2}; S_1^{j_1}$. We thus obtain

$$PI = \langle\, -b[0]*d + b*d[0], a*d[0] - a[0]*d[0] - 2*b[0]*y + 2*b[0]*y[0],$$
$$a*d - a[0]*d - 2*b*y + 2*b*y[0]\rangle.$$

(ii) What remains is to identify the relationship between the polynomial invariants among the loop variables X of the *outer loop* and the computed polynomial relations using Algorithm 4.3 for *an arbitrary iteration of the outer loop*. For doing so, we proceed as follows.

1. Note that the initial values X_0 of the loop variables X at the entry point of the outer loop are also the initial values of the loop variables X before the *first* iteration of the outer loop (8). We thus firstly compute by Algorithm 4.3 the ideal of all polynomial relations among the loop variables X with initial values X_0 corresponding to the *first* iteration of the outer loop (8). We denote this ideal by PI_1.
2. Next, from (the generators of) PI_1 we keep only the set GI of polynomial relations that are invariants among the loop variables X with initial values X_0: they are preserved by any iteration of the outer loop (8) starting in a state in which the initial values of the loop variables X are X_0. By correctness of Theorem 4.5, the polynomials from GI thus obtained are invariants among the loop variables X with initial values X_0 of the P-solvable loop (7) (see Theorem 4.10).

Finally, we can now formulate our algorithm for polynomial invariant generation for P-solvable loops with conditional branches and assignments.

Algorithm 4.4 P-solvable Loops with Non-deterministic Conditionals
Input: P-solvable loop (7) with $k \geq 1$ conditional branches and assignments
Output: Polynomial invariants of (7) among X with initial values X_0

1 Apply Theorem 4.5, yielding a nested loop (8) with k P-solvable inner loops $S_1, \ldots, S_k$
2 Apply Algorithm 4.3 for computing the ideal PI_1 of polynomial relations among X after the first iteration of the outer loop (8)
3 From PI_1 keep the set GI of those polynomials whose conjunction is preserved by each $S_1, \ldots, S_k$:

$GI = \{p \in PI_1 \mid wp(S_i, p(X) = 0) \in \langle GI\rangle, i = 1, \ldots, k\} \subset PI_1$, where $wp(S_i, p(X) = 0)$ is the weakest precondition of S_i with postcondition $p(X) = 0$

4 $\underline{return}\ GI$.

Theorem 4.10. Algorithm 4.4 is correct. It returns polynomial invariants among the loop variables X with initial values X_0 of the P-solvable loop (7).

Example 4.11. From Example 4.9 we already have the set PI_1 for Example 4.6. By applying step 3 of Algorithm 4.4, the set of polynomial invariants for Example 4.6 is

$$GI = \{b[0]*d + b*d[0], a*d[0] - a[0]*d[0] - 2*b[0]*y + 2*b[0]*y[0],$$
$$a*d - a[0]*d - 2*b*y + 2*b*y[0]\}.$$

In what follows, we state under which additional assumptions Algorithm 4.4 returns a basis of the polynomial invariant ideal. We fix some further notation.

- J_* the polynomial invariant ideal among X with initial values X_0 of the P-solvable loop (7).
- J_W denotes the ideal of polynomial relations among X with initial values X_0 after S_W^J.
- For all $i = 1, \ldots, k$ and $j \in \mathbb{N}$, we denote by $J_{W,i}$ the ideal of polynomial relations among X with initial values X_0 after $S_W^J; S_i^j$.

For proving completeness of our method, we impose structural conditions on the ideal of polynomial relations among X with initial values X_0 corresponding to sequences of k and $k+1$ inner loops, as presented below.

Theorem 4.12. Let $\mathfrak{a}_k = \bigcap_{W \in \mathfrak{S}_k} J_W$, $\mathfrak{a}_{k+1} = \bigcap_{\substack{W \in \mathfrak{S}_k \\ i=1,\ldots,k}} J_{W,i}$. Let GI be as in Algorithm 4.4.

1. If $\mathfrak{a}_k = \mathfrak{a}_{k+1}$ then $J_* = \mathfrak{a}_k$.
2. If $\langle GI \rangle = \mathfrak{a}_k \cap \mathfrak{a}_{k+1}$ then $J_* = \mathfrak{a}_k \cap \mathfrak{a}_{k+1}$.
3. If $\langle GI \rangle = \mathfrak{a}_k$ then $J_* = \mathfrak{a}_k$.

Example 4.13. From Examples 4.9 and 4.11 we obtain $GI = PI_1$. By Theorem 4.12 we thus derive $GI = J_*$, yielding the completeness of Algorithm 4.4 for Example 4.6.

Further Examples. We have successfully tested our method on a number of interesting number theoretic examples [17], 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. The fifth columns shows whether our method was complete.

5 Conclusion

A framework for generating loop invariants for a family of imperative programs operating on numbers. We give several methods for invariant generation and prove a number of new results showing soundness, and also sometimes completeness of these methods. These results use non-trivial mathematics based on combining combinatorics, algebraic relations and logic. Moreover, the framework is implemented as a *Mathematica* package, called `Aligator`, and used further for imperative program verification in the *Theorema* system. A collection of examples successfully worked out using the framework is presented in [17].

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

Example	Comb. Methods	Nr.Poly.	(sec)	Compl.
P-solvable loops with assignments only				
Division [6]	Gosper	1	0.08	yes
Integer square root [15]	Gosper	2	0.09	yes
Integer square root [16]	Gosper	2	0.09	yes
Integer cubic root [16]	Gosper	2	0.15	yes
Fibonacci [17]	Generating functions, Alg.Dependencies	1	0.73	yes
Sum of powers n^5 [24]	Gosper	1	0.07	yes
P-solvable loops with conditional branches and assignments				
Wensley's Algorithm [27]	Gosper, C-finite, Alg.Dependencies	2	0.48	yes
LCM-GCD computation [6]	Gosper	1	0.33	yes
Extended GCD [16]	Gosper	5	2.65	yes
Fermat's factorization [16]	Gosper	1	0.32	yes
Square root [29]	Gosper, C-finite, Alg.Dependencies	1	1.28	yes
Binary Product [16]	Gosper, C-finite, Alg.Dependencies	1	0.47	yes
Binary Product [25]	Gosper, C-finite, Alg.Dependencies	1	9.6	yes
Binary Division [10]				
1st Loop	C-finite, Alg. Dependencies	2	0.10	yes
2nd Loop	C-finite, Gosper, Alg.Dependencies	1	0.72	yes
Square root [6]				
1st Loop	C-finite, Alg. Dependencies	2	0.15	yes
2nd Loop	Gosper, C-finite, Alg. Dependencies	1	8.7	yes
Hardware Integer Division [20]				
1st Loop	C-finite, Alg. Dependencies	3	0.19	yes
2nd Loop	Gosper, C-finite, Alg.Dependencies	3	0.64	yes
Hardware Integer Division [26]				
1st Loop	C-finite, Alg. Dependencies	3	0.17	yes
2nd Loop	Gosper, C-finite, Alg.Dependencies	3	0.81	yes
Factoring Large Numbers [16]	C-finite, Gosper	1	14.4	yes

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. We are also interested in generalizing the framework to programs on nonnumeric data structures.

Acknowledgements. The author wishes to thank Tudor Jebelean, Andrei Voronkov, Deepak Kapur and Manuel Kauers for their help and comments.

References

1. Buchberger, B.: Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In: Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems, pp. 184–232 (1985)
2. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards Computer-Aided Mathematical Theory Exploration. Journal of Applied Logic 4(4), 470–504 (2006)
3. Collins, G.E.: Quantifier Elimination for the Elementary Theory of Real Closed Fields by Cylindrical Algebraic Decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
4. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 84–97 (1978)
5. Cox, D., Little, J., O'Shea, D.: Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra, 2nd edn. Springer, Heidelberg (1998)

6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
7. Everest, G., van der Poorten, A., Shparlinski, I., Ward, T.: Recurrence Sequences. Mathematical Surveys and Monographs, American Mathematical Society, 104 (2003)
8. German, S.M., Wegbreit, B.: A Synthesizer of Inductive Assertions. IEEE Transactions on Software Engineering 1, 68–75 (1975)
9. Gosper, R.W.: Decision Procedures for Indefinite Hypergeometric Summation. Journal of Symbolic Computation 75, 40–42 (1978)
10. Kaldewaij, A.: Programming. The Derivation of Algorithms. Prentince-Hall, Englewood Cliffs (1990)
11. Kapur, D.: A Quantifier Elimination based Heuristic for Automatically Generating Inductive Assertions for Programs. Journal of Systems Science and Complexity 19(3), 307–330 (2006)
12. Karr, M.: Affine Relationships Among Variables of Programs. Acta Informatica 6, 133–151 (1976)
13. Kauers, M.: SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. Journal of Symbolic Computation 41, 1039–1057 (2006)
14. Kauers, M., Zimmermann, B.: Computing the Algebraic Relations of C-finite Sequences and Multisequences. Technical Report 2006-24, SFB F013 (2006)
15. Kirchner, M.: Program Verification with the Mathematical Software System Theorema. Technical Report 99-16, RISC-Linz, Austria, Diplomaarbeit (1999)
16. Knuth, D.E.: The Art of Computer Programming, 3rd edn. vol. 2. Addison-Wesley, Reading (1998)
17. Kovács, L.: Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema. PhD thesis, RISC, Johannes Kepler University Linz (2007)
18. Kovács, L., Jebelean, T.: Finding Polynomial Invariants for Imperative Loops in the Theorema System. In: Proc. of Verify 2006, FLoC 2006, pp. 52–67 (2006)
19. Kovács, L., Popov, N., Jebelean, T.: Combining Logic and Algebraic Techniques for Program Verification in Theorema. In: Proc. of ISOLA 2006 (2006)
20. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill Inc, New York (1974)
21. Müller-Olm, M., Seidl, H., Petter, M.: Interprocedurally Analyzing Polynomial Identities. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 50–67. Springer, Heidelberg (2006)
22. Müller-Olm, M., Seidl, H.: Polynomial Constants Are Decidable. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 4–19. Springer, Heidelberg (2002)
23. Paule, P., Schorn, M.: A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. Journal of Symbolic Computation 20(5-6), 673–698 (1995)
24. Petter, M.: Berechnung von polynomiellen Invarianten. Master's thesis, Technical University Münich, Germany (2004)
25. Rodriguez-Carbonell, E., Kapur, D.: Generating All Polynomial Invariants in Simple Loops. J. of Symbolic Computation 42(4), 443–476 (2007)
26. Sankaranaryanan, S., Sipma, H.B., Manna, Z.: Non-Linear Loop Invariant Generation using Gröbner Bases. In: Proc. of POPL 2004 (2004)
27. Wegbreit, B.: The Synthesis of Loop Predicates. Communication of the ACM 2(17), 102–112 (1974)
28. Zeilberger, D.: A Holonomic System Approach to Special Functions. Journal of Computational and Applied Mathematics 32, 321–368 (1990)
29. Zuse, K.: The Computer - My Life. Springer, Heidelberg (1993)