Automated Generation of Loop Invariants by Recurrence Solving in *Theorema*

Laura Ildikó Kovács, Tudor Jebelean*

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria,
Institute e–Austria, Timisoara, Romania
{kovacs,jebelean}@risc.uni-linz.ac.at

Abstract. Most of the properties established during program verification are either invariants or depend crucially on invariants. The effectiveness of automated verification of (imperative) programs is therefore sensitive to the ease with which invariants, even trivial ones, can be automatically deduced. We present a method for invariant generation that relies on combinatorial techniques, namely on recurrence solving and variable elimination. The effectiveness of the method is demonstrated on examples.

AMS Subject Classification: 33F10, 65G20, 68N30, 68Q60, 68W30
Keywords and phrases: program analysis, verification, invariant generation, recurrence solving, symbolic computation

1 Introduction

Verification of imperative programs is a cumbersome task, since additional theory is needed to relate program statements to logic, and some of the logic of the program (such as loop invariants) is often not explicitly available. Thus, invariants and intermediate assertions are the key to deductive verification of imperative programs. Correspondingly, techniques for automatically checking and finding invariants and intermediate assertions have been studied (cf., e.g., [13],[8],[3]). The importance of automated invariant generation for verification of a problem is well–known. Invariant assertions (i.e. assertions that are true of any program state reaching that location) can be used to establish properties of a program and to obtain lemmas for proving safety and correctness properties.

In this paper we present our practical approach to program verification using *backward propagation*, most specifically verification of while loops, in the frame of the *Theorema* (`www.theorema.org`) system. Our approach for invariant generation is build upon the *difference equations method* ([1]), which proceeds in two steps: first, by means of recurrence equations (also called *difference equations*),

* 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 Timisoara project.

an explicit expression is found for the value of each variable as a function of the number of the loop iteration k, other variables that remain constant in the loop, and the input variables; then the variable k is eliminated to obtain invariant properties. Our method uses powerful algorithms from combinatorics, namely recurrence solvers and variable elimination. Nevertheless, only using recurrence solvers, one cannot detect all the invariance properties in general. Some properties, such as inequations, non–linear constraints, etc., might be also necessary for the verification process. For this purpose, one can analyze the postcondition of a loop to extract relevant information. A more powerful and abstract technique was presented in [6], where non–linear constraint solving and quantifier elimination is used to attack the problem of finding invariant linear inequalities. Our ongoing work is to investigate this approach, together with the applicability of Gröbner bases [13], [8], [3] and elimination theory, which are not as costly as the general methods for quantifier elimination.

The main contribution of this paper consists in combining the combinatorial methods with the automated generation of invariants in the Theorema system, namely the technique of generating functions and solving non–linear recurrences. Furthermore, build upon these techniques, and also the technique of Gosper–solvable recurrences (see [11]), we developed a method that can generate invariants in the case of inner–loops.

2 The Working Environment: *Theorema*

The *Theorema* group is active since 1994 in the area of computer aided mathematics, with main emphasis on developing automated reasoning. The *Theorema* system (`www.theorema.org`) is an integrated environment for mathematical explorations [4]. In particular, the *Theorema* system offers support for computing, proving and solving mathematical expressions using specified knowledge bases, by applying several simplifiers, solvers and provers, which imitate the style used by human when proving mathematical statements. The *Theorema* system tries to combine proving, computing, and solving, use of computer algebra, special sequent calculus, domain specific provers, induction, use of meta–variables, etc.

Moreover, *Theorema* offers the possibility of composing, structuring and manipulating arbitrary complex mathematical texts consisting of formal mathematical expressions together with structural information like labels or keywords such as Definition, Theorem, Proposition, Algorithm, etc.

Algorithms can be expressed in *Theorema* using the language of predicate logic, with equalities interpreted as rewrite rules (which leads to an elegant functional programming style), and program verification is done by proving specifications based on definitions (both are logical formulae). However, the system also contains an imperative language with interpreter and verifier, allowing program verification for imperative programs by generating and proving verification conditions depending on the program text [9].

The *Theorema* system is particularly appropriate for program verification, because it delivers the proofs in a natural language by using natural style inferences. The system is implemented on top of the computer algebra system *Mathematica* [18], thus it has access to a wealth of powerful computing and solving algorithms.

3 Application of Invariant Generation to Program Verification

Programs written in functional style can be expressed directly in the *Theorema* language, thus the "compilation" step (and its possible errors) is avoided. However, for users which are more comfortable with the imperative style, *Theorema* features a procedural language, as well as a verification condition generator based on a practically oriented version of the theoretical frame of Hoare–Logic, namely on the Weakest Precondition Strategy [9], [2]. This verification tool provides readable arguments for the correctness of programs, with useful hints for debugging.

The user interface has few simple and intuitive commands ($Program$, $Specification$, VCG, $Execute$). The programs are considered as procedures, without return values and with input, output and/or transient parameters. The programming language features integer variables, the usual arithmetic operations ($+$, $*$, div, mod, etc.). Programs are annotated with pre- and postcondition, loop invariants and termination terms (invariants involving linear inequalities and modulo operations still have to be given by the users). The source code of a program contains a sequence of the following statements: [9, 15]:

- assignments (may contain also function calls);
- conditional statements: IF[*cond*, *THEN-branch*, *ELSE-branch*]
- WHILE loops: $WHILE$[*cond*, *body*, optional:$Invariant, TerminationTerm$]
- FOR loops: FOR[*counter*, *lowerBound*, *upperBound*, *step*, *body*, $Invariant$];
- procedure calls

In this imperative language, for the WHILE and FOR statements we allow additional arguments, namely the invariant and termination term in the case of WHILE loop, and invariant in the case of FOR loop. These optional arguments are relevant in the verification process of the program. A restriction on the body of a loop is that it may contain only assignments or other loop, no $If - Then - Else$ statements.

Basically the verification of programs contains three components.

First of all, the Verification Condition Generator (VCG), that takes an annotated program with pre– and postcondition (i.e. the program's specification), and produces, as output, a verification condition containing a collection of formulas that must be satisfied in order to ensure the (partial) correctness of the program. The verification condition generator is based on a list of inference rules. It is recursive on the structure of the code, and works back–to–front, statement by statement. The conditions are generated according to Floyd–Hoare–Dijkstra's inductive assertion method [7], by applying the weakest precondition strategy, so that each verification condition is associated to a fragment of code. Thus, internally, VCG repeatedly modifies the postcondition using a predicate transformer, such that at the end the result is a list of verification conditions in the *Theorema* syntax. The invariant (and termination term) generation is performed in this phase.

Subsequently, as a second part of the verification, the generated verification conditions are given to the automated theorem provers of the *Theorema* system in order to check whether they hold. The obtained proofs are generated using natural style of inferences, and they return as final output $Proved$ for successfully proved situations or $Pending$ for proof failures.

Finally, a third component of our verification process is the proof–analyzer. This is still an ongoing work. The analyzer would be applied for pending proofs, in order to retrieve useful information about why and where the proof got stopped. Thus, we would be able to obtain additional knowledge, which, proven separately, can be embedded in the theory that is necessary for proving correctness.

The invariants obtained with our combinatorial approach are used to prove automatically the (partial) correctness of programs. We performed and succeeded with this proving process for a number of various examples. Our verifier is still under development in the *Theorema* system.

4 Generation of Loop Invariants

Verification of correctness of loops needs additional information, so-called annotations. In the case of FOR loops these annotations consist in the invariants only, but in the case of $WHILE$ loops, beside the invariant, another annotation is a termination term necessary for proving termination [10].

In most verification systems, these annotations are given by the user. It is generally agreed [5] that finding automatically such annotations is difficult. However, in most of the practical situations, finding invariant expression – or at least giving some useful hints – is quite feasible. For practical applications this may be very helpful for the user. In this paper we present our work–in–progress technique for automated invariant generation by combinatorial and algebraic methods. So far, in the invariant generation process, we dealt with three type of recursive equations, namely with Gosper summable, non–linear and mutual recurrences, each of them being presented in the next subsections.

Let us denote by X the set of (global) variables the program operates on; we assume that the variables take values in the fixed field of real numbers $\mathbb{R}$. Also, for our technique, we assume that the statements from the body of a loop are polynomial assignments of the form $x := p$ (where $x \in X$ and $p \in \mathbb{R}[X]$) or While loops. Moreover, positive Boolean combinations can be also treated, as follows:

- conjunction of polynomial assignments is considered as a sequence of separated assignments
- manipulation of disjunction of polynomial assignments is done by using product properties, i.e. $p_1 = 0 \bigvee p_2 = 0$ is rewritten as $p_1 \cdot p_2 = 0$

4.1 Solving First Order Recurrences

Gosper summable Reccurences Analyzing the code of a loop, we can generate recursive equations for those variables which are modified during the execution of the loop (called *critical* variables). By means of these recurrence equations (also called called difference equations), explicit expression is found for the value of each critical variable as a function of the number of loop iteration, other variables that remain constant in the loop and the input variables. Afterwards, we eliminate the variable which refers to the current iteration of the loop to obtain invariant formulas. Thus, the resulting equation(s) contain the information that have to embedded in the invariant of the loop. This generation process was presented in much more detail in [11]. For illustration, consider the "Division" program of two natural numbers:

$$
\begin{array}{l}
Specification[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],\\
\quad Pre \rightarrow ((x \geq 0) \wedge (y > 0)),\\
\quad Post \rightarrow ((quo * y + rem = x) \wedge (0 \leq rem < y))]\\
Program[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],\\
\quad quo := 0;\\
\quad rem := x;\\
\quad WHILE[y \leq rem,\\
\qquad rem := rem - y;\\
\qquad quo := quo + 1]
\end{array}
$$

By our approach, we are able to generate the invariant equation:

$$rem = x - quo * y.$$

By, further analysis of conditions upon the the variables of the loop–body, we generate the final invariant:

$$Invariant \equiv (quo * y + rem = x) \wedge 0 \leq rem$$

Non–linear Recurrences When the obtained recurrence is not Gosper–summable, but it is of the form:

$$x_n = t * x_{n-1}^c,$$

where t is a term that does not depend on n, and $c \in \mathbb{Q}$, the closed form of the detected recurrence can be solved by our recurrence solving package.

The presented strategy works only if we do not have mutual recurrence(s) in the loop body.

4.2 Mutual Recurrences

In most cases, beside the above recurrences, a program–code contains higher order recurrences. An interesting case of this type of recurrence is the case of *mutual* recurrence, where, for instance, two equations are mutually depending on each other. For solving such a problem, we use the technique of *generating functions* [17] from combinatorics ([17] is an original work about P-finite and D-finite sequences, namely about closure properties).

Consider the example of the well-known program for computing the Fibonacci numbers, which has its specification and source code, as follows:

$$
\begin{array}{l}
Specification["Fibonacci",\ FibonacciProcSpec[\downarrow n, \uparrow F],\\
\quad Pre \rightarrow (n \geq 0),\\
\quad Post \rightarrow (F = FibExp[n])]\\
Program["Fibonacci",\ FibonacciProc[\downarrow n, \uparrow F],\\
\quad Module[\{H,\ i\},\\
\quad\quad i\ :=\ n;\\
\quad\quad F\ :=\ 1;\\
\quad\quad H\ :=\ 1;\\
\quad\quad WHILE[i\ >\ 1,\\
\quad\quad\quad H\ :=\ H\ +\ F;\\
\quad\quad\quad F\ :=\ H\ -\ F;\\
\quad\quad\quad i\ :=\ i\ -\ 1]]
\end{array}
$$

Note: $FibExp[n]$ denotes the term: $F_k = \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi}$ is its conjugate.

From the loop–body, we can merely set up the recurrence:

$$H_k\ =\ H_{k-1}\ +\ F_{k-1}\ (k \geq 1),\ H_1 = 1$$

$$F_k\ =\ H_k\ -\ F_{k-1}\ (k \geq 1),\ F_1 = 1$$

For solving this problem, we apply the technique of *generating functions*, namely, given a sequence (g_k) that satisfies a given recurrence (in our example, the sequences are (F_k) and (H_k), we seek a closed form for g_k in term of k.

For our computations we extend Mallinger's Mathematica package GeneratingFunctions [12], which was developed in the Combinatoric Group of RISC. In order to apply this package–extension, first we rewrite the equations of expressions H_k and F_k in such a way that they are valid for all integers k, assuming that $H_0 = H_{negative} = 0$ and $F_0 = F_{negative} = 0$.

Hence we obtain:

$$H_k\ =\ H_{k-1}\ +\ F_{k-1}\ +\ [k = 1]\ (k \in Z)$$

$$F_k\ =\ H_k\ +\ F_{k-1}\ (k \in Z)$$

(where the meaning of $[k = 1]$ is that it adds 1 (i.e. H_1) when $k = 1$, and it makes no change when $k \neq 1$.

Then, by applying the *generating functions* technique, we obtain the harmonic forms:

$$
\begin{aligned}
H(z) &= \sum_k H_k z^k = \sum_k H_{k-1} z^k + \sum_k F_{k-1} z^k + \sum_k [k=1] z^k \ (k \in Z) \\
&= zH(z) + zF(z) + z \\
F(z) &= \sum_k H_k z^k + \sum_k F_{k-1} z^k \ (k \in Z) \\
&= H(z) + zF(z)
\end{aligned}
$$

Solving a system of two equations with two unknowns, we obtain the generating functions:

$$
\begin{aligned}
F(z) &= \frac{z}{1-z-z^2} \\
H(z) &= \frac{z(1+z)}{1-z-z^2}
\end{aligned}
$$

and thus, by combinatorial techniques, the desired closed form of their coefficients is determined, namely:

$$
\begin{aligned}
F_k &= \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}} \\
H_k &= \frac{\phi^{k+1} - \hat{\phi}^{k+1}}{\sqrt{5}}
\end{aligned}
$$

Hence, at the k^{th} iteration, we have as invariance properties:

$$
\begin{aligned}
F_k &= \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}} \\
H_k &= \frac{\phi^{k+1} - \hat{\phi}^{k+1}}{\sqrt{5}}
\end{aligned}
$$

From the third recurrence equation of the loop, i.e. $i_{k+1} = i_k - 1, i_0 = n$, by the Gosper algorithm, we obtain the closed form: $i_k = n - (k-1)$.

In the following steps, we proceed as in the previous section, namely we eliminate the loop's counter variable k from the three equations, thus we obtain:

$$
\left(F = \frac{\phi^{n-i+1} - \hat{\phi}^{n-i+1}}{\sqrt{5}} \right) \bigwedge \left(H = \frac{\phi^{n-i+2} - \hat{\phi}^{n-i+2}}{\sqrt{5}} \right)
$$

One notes that these are exactly the expressions of the Fibonacci numbers [14].

Summarizing, the technique of *generating functions* turns out to be practical and effective for solving higher order recurrences, thus for generating invariance properties for more complex programs.

The theoretical details of these methods are described in [16].

5 Invariant Generation for WHILE loops with WHILE statements

So far in the automatic invariant generation we dealt with cases when a loop had only assignments. However, more interesting situations arise in the case of nested whiles, i.e. with while loops in a body of a while loop. Therefore, the next step in our work is to develop an invariant generation method for nested whiles. There are other approaches for doing so [13], [8], namely a method built upon polynomial ideal theory, using Gröbner bases. By their approach, the invariant generation problem is translated to a (linear or non–linear) constraint solving problem (where the constraints describe properties of the coefficients of the polynomial loop invariant).

In our method, as a continuation of the combinatorial approach, the main idea is to simulate the execution of whiles with recurrence equations. We have seen, that in the case of loops with only assignment statements the combinatorial manipulation of the statements from the loop's body is efficient for invariant generation. Starting from this base–case, as a first step, we annotate our programs in such a way that the combinatorial approach can be performed.

There are several ways for modelling programs, e.g. in [12] they use (algebraic) transition systems (with set of variables, locations, transitions between locations, an initial condition and assertion), while in [8] the notion of control flow–graphs is introduced and used (with set of program points, set of edges between points, a mapping that annotates each edge with an assignment statements and with a start point). In our approach, in order to succeed with the combinatorial approach, we have to take special care of the counters of the loops, i.e. the loop–iterations. Doing so, the first step of our invariant generation problem will consist of building the set of loop–variables V and assigning counters to loops w.r.t. the hierarchy they appear.

For a better understanding, we illustrate this step for the *Binary Power* problem, having its source code and specification as follows:

$$\begin{array}{l}
\textbf{Specification}[\text{"}BinaryPower\text{"},\ BinPowerSpec[\downarrow x,\ \downarrow y,\ \uparrow z],\\
\quad Pre \rightarrow \left((y \geq 0) \bigwedge (IsInteger[y])\right),\\
\quad Post \rightarrow (z\ =\ x^y)\ ],\\
\textbf{Program}[\text{"}BinaryPower\text{"},\ BinPower[\downarrow x,\ \downarrow y,\ \uparrow z],\\
\qquad Module[\{a, b\},\\
\qquad\quad z := 1;\\
\qquad\quad a := x;\\
\qquad\quad b := y;\\
\qquad\quad WHILE[b > 0,\\
\qquad\qquad b := b - 1;\\
\qquad\qquad z := z * a;
\end{array}$$

$$
\begin{aligned}
&\qquad WHILE[\ IsEven[b]\ \wedge\ b > 0,\\
&\qquad\quad a := a * a;\\
&\qquad\quad b := b/2,\\
&\qquad\quad b := b/2\\
&\qquad]\,]\\
&](*Module*)\ ].
\end{aligned}
$$

Thus, $V = \{a, b, z\}$ for both loops, and we will have the following counter–assignments:

$$
\begin{aligned}
j_0 :\quad & WHILE[b > 0,\\
& \qquad b := b - 1;\\
& \qquad z := z * a;\\
& \qquad j_0^1 :\quad WHILE[\ IsEven[b]\ \wedge\ b > 0,\\
& \qquad\qquad a := a * a;\\
& \qquad\qquad b := b/2]]
\end{aligned}
$$

These annotations have intuitively the meaning, that the loop with counter j_0^1 is the first sub–loop of the loop with the counter j_0.

The next step of the generation process is the statement– and variable–manipulation. Therefore, we proceed as follows:

- split the body of the outer–loop into separated parts: one part will contain the assignment statements that are present before the inner while, the next part the inner while, the third part the assignment statements after the inner while (and if other while are also present, the splitting continues)
- for each part, rewrite the recursive polynomial assignment using the proper indexes (loop–counters). For those variables that do not change in the specific part, consider the assignment that describes the constant property of them (i.e. $x_{j+1} := x_j$)
- for the inner while, by the combinatorial methods for summation, generate a polynomial equation(s) that contain invariant properties
- replace the inner while loop with its (polynomial) invariant(s), taking care of the proper specification of the loop counter, and that the initial values of the variables of the inner loop are given by the outer loop's variables.
- solve – by repeated substitutions – the obtained system of recursive equations, obtaining the invariant for the outer loop. In this process, also some other properties can be established that describe the initial values of the critical variables of the inner loop.

For a better understanding, let us continue the example of the *Binary Power*. In this case we proceed as follows:

- For the inner loop, we obtain, by recurrence solving:

$$
b_{j_0^1} = \frac{b_{j_0^1{}_{ini}}}{2^{j_0^1}} \qquad a_{j_0^1} = a_{j_0^1{}_{ini}}^{2^{j_0^1}}
$$

where $a_{j^1_{0\,ini}}$ and $b_{j^1_{0\,ini}}$ are the values of a and b before the inner loop. Hence, (by special solving and elimination techniques for j^1_0, the invariant is:

$$a_{j^1_0}^{b_{j^1_0}} = a_{j^1_{0\,ini}}^{b_{j^1_{0\,ini}}}$$

and

$$z_{j^1_0} = z_{j^1_{0\,ini}}$$

– Referring to the counter of the outer–loop, we will have the substitution in the above invariant:

$$a_{j^1_0} \to a_{j_0+1};\quad b_{j^1_0} \to b_{j_0+1};\quad a_{j^1_{0\,ini}} \to a_{j_0};\quad b_{j^1_{0\,ini}} \to b_{j_0};\quad z_{j^1_0} \to z_{j_0+1};\quad z_{j^1_{0\,ini}} \to z_{j_0}$$

and obtain the system of equations:

$$\left\{\begin{array}{ll} b_{j_0} = b_{j_0-1} - 1 & (1) \\ z_{j_0} = z_{j_0-1} * a_{j_0-1} & (2) \\ a_{j_0} = a_{j_0-1} & (3) \\ a_{j_0+1}^{b_{j_0+1}} = a_{j_0}^{b_{j_0}} & (4) \\ z_{j_0+1} = z_{j_0} & (5) \end{array}\right\}$$

which, by bottom–up substitutions leads to the system:

$$\left\{\begin{array}{ll} a_{j_0+1}^{b_{j_0+1}} = a_{j_0-1}^{b_{j_0-1}-1} & (6) \\ z_{j_0+1} = z_{j_0-1} * a_{j_0-1} & (7) \end{array}\right.$$

which leads to the product formula:

$$z_{j_0+1} * a_{j_0+1}^{b_{j_0+1}} = z_{j_0-1} * a_{j_0-1}^{b_{j_0-1}},$$

thus, by initial values replacements, we obtain the invariant:

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

In the case when a loop contains two or more inner loops, our technique is still applicable. For illustration, consider the program (and its specification) that calculates simultaneously the lcm and the gcd of two integers x_1 and x_2.

Specification["$LCM - GCD$", LcmGcdSpec[$\downarrow x1,\ \downarrow x2,\ \uparrow y1,\ \uparrow y3,\ \uparrow y4$],
$\quad Pre \to IsInteger[x1, x2, y1, y3, y4]$,
$\quad Post \to (y1 = \ GCD[x1, x2]) \wedge y3 + y4 = LCM[x1, x2]]$,

(where LCM and GCD are the *Mathematica* built–in functions that computes the LCM and GCD of two integers.)

Program["$LCM - GCD$", LcmGcd[$\downarrow x1,\ \downarrow x2,\ \uparrow y1,\ \uparrow y3,\ \uparrow y4$],
$\qquad Module[\{y2\}$,

$y1 := x1;$
$y2 := x2;$
$y3 := x2;$
$y4 := 0;$
$WHILE[y1 \neq y2,$
$\quad WHILE[\ y1 > y2,$
$\qquad y1 := y1 - y2;$
$\qquad y4 := y4 + y3$
$\quad];$
$\quad WHILE[\ y2 > y1,$
$\qquad y2 := y2 - y1;$
$\qquad y3 := y3 + y4$
$\quad]]$
$](*Module*)]$

For generating the invariant, we build the set of variables of each loop and start by assigning to each loop a counter. In this example, the set of variables is common, e.g. $V = \{x1, x2, y1, y3, y3, y4\}$ and we obtain the following annotated code:

$j_0 :\ WHILE[y1 \neq y2,$
$\quad j_0^1 :\ WHILE[\ y1 > y2,$
$\qquad y1 := y1 - y2;$
$\qquad y4 := y4 + y3];$
$\quad j_0^2 :\ WHILE[\ y2 > y1,$
$\qquad y2 := y2 - y1;$
$\qquad y3 := y3 + y4$
$\qquad]]$

Hence, for each inner–loop, we obtain two systems of recursive equations which by recurrence solving lead to the system of equations:

$$\text{first inner loop} \left\{ \begin{array}{ll} y1_{j_0^1} := y1_{j_0^1{}_{ini}} - j_0^1 * y2_{j_0^1} & (1.1) \\ y4_{j_0^1} := y4_{j_0^1{}_{ini}} + j_0^1 * y3_{j_0^1} & (1.2) \\ y2_{j_0^1} := y2_{j_0^1{}_{ini}} & (1.3) \\ y3_{j_0^1} := y3_{j_0^1{}_{ini}} & (1.4) \end{array} \right\}$$

and

$$\text{second inner loop} \left\{ \begin{array}{ll} y2_{j_0^2} := y2_{j_0^2{}_{ini}} - j_0^2 * y1_{j_0^2} & (2.1) \\ y3_{j_0^2} := y3_{j_0^2{}_{ini}} + j_0^2 * y4_{j_0^2} & (2.2) \\ y1_{j_0^2} := y1_{j_0^2{}_{ini}} & (2.3) \\ y4_{j_0^2} := y4_{j_0^2{}_{ini}} & (2.4) \end{array} \right\}$$

Thus, by elimination of j_0^1 and j_0^2, we obtain the following invariants:

$$\text{first inner loop:} \left\{ \begin{array}{ll} y1_{j_0^1} * y3_{j_0^1} + y2_{j_0^1} * y4_{j_0^1} := y3_{j_0^1} * y1_{j^1_{0\,ini}} + y2_{j_0^1} * y4_{j^2_{0\,ini}} & (1.5) \\ y2_{j_0^1} := y2_{j^1_{0\,ini}} & (1.6) \\ y3_{j_0^1} := y3_{j^1_{0\,ini}} & (1.7) \end{array} \right\}$$

$$\text{second inner loop:} \left\{ \begin{array}{ll} y2_{j_0^2} * y4_{j_0^2} + y3_{j_0^2} * y1_{j_0^2} := y2_{j^2_{0\,ini}} * y4_{j_0^2} + y1_{j_0^2} * y3_{j^2_{0\,ini}} & (2.5) \\ y1_{j_0^2} := y1_{j^2_{0\,ini}} & (2.6) \\ y4_{j_0^2} := y4_{j^2_{0\,ini}} & (2.7) \end{array} \right\}$$

and the counter–correspondence for variables:

$$\begin{pmatrix} y1_{j^1_{0\,ini}} \to y1_{j_0} \\ y2_{j^1_{0\,ini}} \to y2_{j_0} \\ y3_{j^1_{0\,ini}} \to y3_{j_0} \\ y4_{j^1_{0\,ini}} \to y4_{j_0} \\ y1_{j_0^1} \to y1_{j_0+1} \\ y2_{j_0^1} \to y2_{j_0+1} \\ y3_{j_0^1} \to y3_{j_0+1} \\ y4_{j_0^1} \to y4_{j_0+1} \end{pmatrix} \begin{pmatrix} y1_{j^2_{0\,ini}} \to y1_{j_0^1} \\ y2_{j^2_{0\,ini}} \to y2_{j_0^1} \\ y3_{j^2_{0\,ini}} \to y3_{j_0^1} \\ y4_{j^2_{0\,ini}} \to y4_{j_0^1} \\ y1_{j_0^2} \to y1_{j_0^1+1} \\ y2_{j_0^2} \to y2_{j_0^1+1} \\ y3_{j_0^2} \to y3_{j_0^1+1} \\ y4_{j_0^2} \to y4_{j_0^1+1} \end{pmatrix}$$

Hence, by backward substitution (of (2.6) and (2.7) into (2.5)), and index manipulation, we obtain first:

$$\text{second inner loop:} \quad y2_{j_0^1+1} * y4_{j_0^1} + y3_{j_0^1+1} * y1_{j_0^1} := y2_{j_0^1} * y4_{j_0^1} + y1_{j_0^1} * y3_{j_0^1}$$

which, using the invariant equations and counter–manipulations of the first inner–loop, yields the main invariant equation:

$$\text{first inner loop:} \quad y2_{j_0+2} * y4_{j_0+2} + y3_{j_0+2} * y1_{j_0+2} := y2_{j_0} * y4_{j_0} + y1_{j_0} * y3_{j_0}),$$

which, by initial value substitution, yields the invariant:

$$y1 * y3 + y2 * y4 = x1 * x2.$$

(Note: For the verification process, applying the exit condition $y1 = y2$ yields, $y1 * (y3 + y4) = x1 * x2$. Assuming that $y1 = \mathrm{GCD}(x1, x2)$ and $y3 + y4 = \mathrm{LCM}(x1, x2)$, the invariant states that $\mathrm{LCM}(x1, x2) * \mathrm{GCD}(x1, x2) = x1 * x2$. Note that correctness cannot be inferred directly since LCM and GCD functions cannot be expressed algebraically. For correctness additional knowledge is necessary.)

We have tried our method with several examples, and succeeded to generate in each case the loop invariant. Currently we are working on the theoretical basis and justification of this approach, in order to ensure correctness (and completeness) of the method.

6 Generation of Termination Terms

In the case of the WHILE loop, one is also interested to be able to prove termination, i.e. to have an automatic generation of a termination term. Knowing that the termination term must be positive [7], we transform the given loop-condition Φ using specific heuristics (algebraic manipulations) until we obtain a term T such that $T \geq 0 \Leftrightarrow \Phi$.

In the *Division* example, the termination term will be:

$$rem - y$$

7 Conclusions and Further Work

Combined with a practically oriented version of the theoretical frame of Hoare-Logic, *Theorema* provides readable arguments for the correctness of programs, as well as useful hints for debugging. Moreover, it is apparent that the use of algebraic computations (summation methods, variable elimination) is a promising approach to analysis of loops.

Regarding the verifier, our plans in the near future are following:

- implement the proof–analyzer;
- perform additional type checking, since by the Hoare–verification–rule for assignments the replaced value is subsumed to be of the same type as the original value. This, in practice might lead to a situation, when the correctness proof will succeed, although the program is not correct (i.e. the division problem of integers subsumes integer division);
- develop and integrate in the verifier a technique for mechanically inferring loop invariants that are linear inequalities (see [6]);
- enrich the invariant generation technique for the case when a loop–body contains $If - Then - Else$ statements (see [13], [3]).

Another necessary continuation of our work is the analysis of programs containing recursive calls. We are currently investigating the theoretical framework and we are designing the methods for extracting the verification conditions of this type of programs.

References

1. B.Elspas and M.W.Green and K.N.Lewitt and R.J.Waldinger. *Research in Interactive Program–Proving Techniques.* Technical report, Standford Research Institute, Menlo Park, California, USA. Technical Report, May 1972.
2. E. W. Dijkstra. *A Discipline of Programming.* Prentince-Hall, 1976.
3. E.Rodriguez-Carbonell and D.Kapur. *Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations.* In *Proc. of the International Symphosium on Symbolic and Algebraic Computation (ISSAC 04)*, 2004. July 4-7, University of Cantabria, Santander, Spain.

4. B. Buchberger et al. *The Theorema Project: A Progress Report.* In M. Kerber and M. Kohlhase, editors, *Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning.* A. K. Peters, Natick, Massatchussets, 2000.
5. G. Futschek. *Programmentwicklung und Verifikation.* Springer, 1989.
6. M.Colòn and S.Sankaranarayanan and H.B.Sipma. *Linear Invariant Generation Using Non–Linear Constraint Solving.* In *Computer Aided Verification (CAV 2003)*, 2003. vol. 2725 of *Lecture Notes in Computer Science*, pp. 420–432. Springer Verlag.
7. C. A. R. Hoare. *An axiomatic basis for computer programming. Comm. ACM*, 12, 1969.
8. M.Müller-Olm and H.Seidl. *Polynomial Constants are Decidable.* In *Static Analysis Symposium (SAS 2002), vol.2477 of LNCS*, 2002. pp. 4-19.
9. M. Kirchner. *Program verification with the mathematical software system Theorema.* Technical Report 99-16, RISC-Linz, Austria, 1999. PhD Thesis.
10. L. Kovács. *Program Verification using Hoare Logic.* In *Computer Aided Verification of Information Systemsm Romanian-Austrian Workshop*, 2003. Timisoara, Romania, February 2003.
11. L.Kovács and T.Jebelean. *Generation of Invariants in Theorema.* 2003. In N.Boja(ed): Proceedings of the 10th International Symphosium of Mathematics and its Application, Scientific Bulletins of the "Politehnica" University of Timisoara, Romania, Transactions on Mathematics and Physics, ISSN 1224-6069. Timisoara, Romania, November 2003.
12. C. Mallinger. Algorithmic manipulations and transformations of univariate holonomic functions and sequences. Master's thesis, RISC, J. Kepler University, Linz, August 1996.
13. S.Sankaranaryanan and B.S.Henry and Z. Manna. *Non-Linear Loop Invariant Generation using Gröbner Bases.* In *ACM Principles of Programming Languages (POPL'04)*, 2004. January 14-16, Venice, Italy.
14. R.L. Graham and D.E. Knuth and O. Patashnik. *Concrete Mathematics, 2nd ed.* Addison-Wesley Publishing Company, 1989. pg. 306-330.
15. L. Kovács and N. Popov. *Procedural Program Verification in Theorema.* In *Omega-Theorema Workshop*, May 2003. Hagenberg, Austria.
16. B. Salvy and P. Zimmermann. Gfun: A package for the manipulation of generating and holonomic functions in one variable. *ACM Trans. Math. Software*, 20:163–177, 1994.
17. R.P. Stanley. Differentiably finite power series. 1:175–188, 1980.
18. S. Wolfram. *The Mathematica Book, 3rd ed.* Wolfram Media / Cambridge University Press, 1996.