International Journal on Software Tools Technology Transfer manuscript No.
(will be inserted by the editor)

Tudor Jebelean · Laura Kovács · Nikolaj Popov

Experimental Program Verification in the Theorema System

Abstract We describe practical experiments of program verification in the frame of the *Theorema* system. This includes both imperative programs (using Hoare logic), as well as functional programs (using fixpoint theory). For a certain class of imperative programs we are able to generate automatically the loop invariants and then verification conditions, by using combinatorial and algebraic techniques. Verification conditions for functional recursive programs are derived and soundness theorem is proven. The verification conditions in both cases are generated as natural-style predicate logic formulae, which can be then proven by *Theorema*, by issuing natural-style proofs which are human–readable.

Keywords Program Verification · Invariant Generation · Theorem Proving

1 Introduction

We describe the theoretical basis and practical experiments of program verification in the frame of the *Theorema* system (see [4] and also `www.theorema.org`, which has links to our papers cited below). This work (in progress) is built on previous theoretical results and practical experiments regarding verification of functional programs, as well as of imperative programs [3,28,21]. We follow two main approaches:

- Hoare logic for imperative programs,
- fixpoint theory of functions for functional programs.

The first approach consists in using Hoare logic and the weakest precondition method. This is relatively straightforward, however we demonstrate on a certain class of examples how to automatically generate the loop invariants and the termination terms, by applying algebraic and combinatorial techniques.

The second approach is based on the extraction (for certain classes of programs) of the purely logical conditions which are sufficient for the program correctness. These are inferred using Scott induction or induction on natural numbers in the fixpoint theory of functions and constitute a meta-theorem which is proven once for the whole class. The concrete verification conditions for each program are then provable without having to use the fixpoint theory.

These techniques are implemented in the frame of the *Theorema* system and all the examples presented in this paper are treated completely automatically. Currently we are comparing the results of these two approaches, in order to realize a practical verification engine in the frame of the *Theorema* system.

The *Theorema* system is a computer mathematical assistant which is implemented on top of the computer algebra system *Mathematica* [34]. The system supports the activities of defining and organizing mathematical theories (including the description of algorithms) in the language of higher–order predicate logic, and also offers the necessary environment for experimenting with algorithms (computing) and for investigating the properties of mathematical structures (proving, solving). Currently the computing and solving capabilities are imported from the underlying computer algebra system *Mathematica*, and are quite powerful, thus the most significant and interesting part of the *Theorema* system is the one providing proving capabilities. Proving is implemented in the system by way of various domain–specific provers (propositional, first–order predicate logic, limit domains, induction of natural numbers and over lists, proving over sets, proving equalities by Knuth–Bendix

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. The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project F1302.

Tudor Jebelean · Laura Kovács · Nikolaj Popov
Research Institute for Symbolic Computation,
Johannes Kepler University,
A-4040 Linz, Austria
Institute e-Austria Timişoara, Romania
Tel.: +43-732-2468 9946
Fax: +43-732-2468 9930
E-mail: {jebelean,kovacs,popov}@risc.uni-linz.ac.at

completion, etc.). The general approach is to implement inference rules, as well as strategies and techniques which are similar to the human style of proving. The proofs which are produced are explained in natural language and are quite human–readable. Therefore, even failed proofs are useful because they may give hints for finding errors or omissions in the respective theory. Although the main emphasis of Theorema is on fully automatic proving, the system also has interactive facilities for allowing the user to manually influence the behavior of the provers.

The problem of producing the proofs of the verification conditions is not in the scope of this paper. We note, however, that the concrete proof problems issued from program verification examples are used as test cases for the provers of *Theorema* and for experimenting with the organization and management of the mathematical knowledge. Currently we are able to generate the verification conditions and to prove them automatically (in matter of seconds) in the *Theorema* system for many concrete programs. There are, however, more complex examples which take a long time to prove or which cannot be proven fully automatically, but only with user guidance, as there are problems which will require the improvement of the current provers or the design of new provers.

The purpose of our research is to provide an integrated system in which program verification can be done (as much as possible) automatically on many concrete examples. Our approach is incremental and experimental: starting from simpler examples, we improve our methods such that more and more interesting problems can be solved.

We believe that this approach has the potential of increasing the acceptance of formal methods in industry, because both practitioners, but also students (which are future practitioners) are exposed to success stories of concrete program verification.

We consider that this type of what is usually called "toy examples" is actually very important for clarifying the theoretical basis and the details of the algorithms, because they emphasize the essential aspects of using formal methods in program verification. The methods demonstrated in our examples have the potential of playing an important role as building blocks of industrial tools for program verification. Moreover, the system as it stands now can already be used for educational and demonstration purposes.

2 Imperative Programs

Programs written in functional style can be expressed directly in the *Theorema* language (for details see [4]), thus the "compilation" step (and its possible errors) is avoided. However, for users which are more comfortable with the imperative style, we have implemented in *Theorema* a procedural language, as well as a verification condition generator [18] based on Hoare–Logic and using the Weakest Precondition Strategy. 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 contains the following constructs:

- assignments (may contain also function calls);
- blocks: MODULE[set of local variables, sequence of statements]
- conditional statements:
  IF[cond, THEN-branch, ELSE-branch]
- WHILE loops:
  WHILE[cond, body,
  optional: Invariant, TerminationTerm]
- FOR loops:
  FOR[counter, lowerBound, upperBound, step, body,
  optional: Invariant]
- procedure calls.

The optional arguments "Invariant" and "TerminationTerm" are needed for the generation of the verification conditions using the weakest precondition method. The Verification Condition Generator, implemented as "VCG", takes a program and its specification (pre– and postcondition) and produces as output a verification condition containing a collection of formulae. 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. Internally, it 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. This process is relatively straightforward and implements a well-known technique [16, 10], thus we will concentrate in this paper on the **challenging** problem of finding loop invariants and termination terms. The automatically generated invariants (and termination terms), together with other non-algebraic invariants (i.e. modulo expression, inequalities, etc.), are finally used to prove correctness by calling the appropriate provers from the *Theorema* systems. We successfully applied this approach to numerous interesting examples (see [20]), some of them being presented also in this paper.

2.1 Generation of Loop Invariants

Verification of correctness of loops needs additional information, so-called annotations (invariants and termination terms). These annotations are the key to deductive verification of imperative programs. Although several techniques have been considered for automated invariant generation (e.g. abstract interpretation [8], constraint solving [7], polynomial algebra [32, 25, 29]), still

in most verification systems these annotations are given by the user. It is generally agreed [11] that finding automatically such annotations is in general very difficult. However, in most of the practical situations finding the expression – or at least giving some useful hints – is quite feasible.

In this paper we present our work–in–progress technique for automated invariant generation by combinatorial and algebraic methods combined with the automated reasoning power of the *Theorema* system, which extends the work presented in [21]. Driven by practical reasons, the invariants that we are able to obtain automatically are equational (mostly polynomial) relations among the program variables. As stated in [24], generating valid polynomial identities have many applications, such as: constant propagation, discovery of symbolic constants, finding definite equalities among variables, etc. Obtaining automatically invariants also have an important field in the problem of loop optimization (see [13]). Our goal is, starting from simple but basic examples, extend and enrich our method to be able to deal with more complex applications.

We assume that the variables take values in a ring of numbers (usually integers, rationals or reals) $\Re$. In this paper the framework for finding invariants is restricted to a class of programs, where all the statements in the body of a loop are either assignments of the form $x := p$ (x is a program variable, p is a polynomial in the program variables), or WHILE loops. The method presented here relies on intraprocedural analysis (i.e. no procedure calls) where branching in loop statements (i.e. loops with conditional statements) are not "allowed".

2.2 Solving First Order Recurrences

Analyzing the code of a loop, we identify recurrence equations for those variables which are modified during the execution of the loop (called *critical* variables). Using these, an explicit expression is found for the value of each critical variable as a function of the index of loop iteration. By eliminating the loop index from the system of equations, one obtains invariant relations among the critical variables, which have to be embedded in the invariant of the loop.

For illustration, consider the program for the integer division of two natural numbers (the arrows show whether the argument is input or output):

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

$$
\begin{array}{l}
Program[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo], \\
\quad MODULE[, \\
\quad\ quo := 0; \\
\quad\ rem := x; \\
\quad\ WHILE[y \leq rem, \\
\quad\quad rem := rem - y; \\
\quad\quad quo := quo + 1]]]
\end{array}
$$

The automated generation of the invariant proceeds as follows:

From the body of the loop, we obtain the following recursive equations:

$$
\begin{array}{l}
quo_0 = 0;\ \ quo_{k+1} = quo_k + 1 \\
rem_0 = x;\ rem_{k+1} = rem_k - y.
\end{array}
$$

where quo and rem are the critical variables and k is the index of the loop. For each recursive equation we use the Gosper-Zeilberger algorithm (see e.g. [19],[14]). Namely, we use the Paule-Schorn implementation in *Mathematica* [26] which is already embedded in the *Theorema* system, in order to produce a closed–form for the expressions of quo_k and rem_k.

$$
\begin{array}{l}
quo_k \ = \ 0 \ + \ k \\
rem_k \ = \ x \ - \ k * y
\end{array}
$$

(Note that in the above example the closed form is straightforward, because the difference of subsequent values does not depend on k. However, the Gosper-Zeilberger algorithm is able to produce the closed form also when the difference is depending on k, like e. g. in more complex examples presented in [20].)

From these equations we eliminate k by calling the appropriate routine from *Mathematica*, and we obtain the equation:

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

Some additional information which should be embedded in the loop invariant is extracted based on the following principle: at the termination of the loop (i.e. when the condition of the loop is falsified), the so–far generated formulae together with the negation of the loop–condition and the assertion that is still needed to be generated have to imply the postcondition of the loop. Thus, by some heuristics and logical manipulation of formulae, the additional assertion is generated, and finally the complete invariant for this example will be:

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

In the case of the WHILE loop, one is also interested in being able to prove termination, i.e. to have an automatic generation of a termination term. Knowing that the termination term must be positive [16], we transform the given loop-condition, denoted by Φ, using specific heuristics (algebraic manipulations) until we obtain

a term T such that $T \geq 0 \Leftrightarrow \Phi$. In the example above, the termination term will be:

$$rem - y.$$

Finally, as the last step of the verification process, the VCG takes the annotated (with the automatically obtained invariant properties and termination term) source code of the "Division" problem, together with its specification, and, using the weakest precondition strategy, produces a *Theorema* lemma, with four, universally quantified proof obligations, namely:

Lemma(Division)
for any: x, y, rem, quo
(WHILE.Inv+Term)
$(rem - x + quo * y = 0) \ \wedge \ 0 \leq rem \ \wedge \ 0 < y \ \wedge$
$y \leq rem \ \wedge \ (T1 = rem) \implies$
$\big(rem - x - y + (1 + quo) * y = 0\big) \ \wedge \ 0 \leq rem - y \ \wedge$
$0 < y \ \wedge \ rem - y < T1$
(WHILE.Final)
$(rem - x + quo * y = 0) \ \wedge \ 0 \leq rem \ \wedge \ 0 < y \ \wedge$
$\neg(y \leq rem) \implies$
$(rem + quo * y = x) \ \wedge \ 0 \leq rem \ \wedge \ rem < y \ \wedge \ 0 < y$
(WHILE.Term)
$(rem - x + quo * y = 0) \ \wedge \ 0 \leq rem \ \wedge \ 0 < y \ \wedge$
$y \leq rem \implies rem \geq 0$
(Init)
$0 \leq x \ \wedge \ 0 < y \implies (0 = 0) \ \wedge \ 0 \leq x \ \wedge \ 0 < y$

The proof of the above lemma is automatically produced by the the PCS prover of the *Theorema* system [5], that uses quantifier elimination. Thus, we proved (total) correctness of the "Division" program.

In the case of a FOR loop, the generation of the loop invariant is done in the same manner, but we use additionally the explicit equation for the counter of the FOR loop:

$$counter_k := counter_0 + k * step.$$

In the above example, the critical variables are independent on each other. When there are dependencies among the critical variables (but not mutual dependencies), our method is still applicable. In this situation, we work first with that critical variable which does not depend on other ones, we generate the closed–form of it, and then substitute this expression in the other recursive equations. Proceeding in a similar manner for the other recursive equations, we will solve again first–order recursive equations by the Gosper–algorithm.

2.3 Mutual Recurrences

The technique presented above does not work if the loop body contains mutually dependent variables. In this case we use the technique of *generating functions* from combinatorics [33, 31].

We demonstrate it on a concrete example:

$Specification["Fibonacci",\ Fibonacci[\downarrow n, \uparrow F],$
$\quad Pre \rightarrow (n \geq 0),$
$\quad Post \rightarrow (F = FibExp[n])]$

$Program["Fibonacci",\ Fibonacci[\downarrow n, \uparrow F],$
$\quad MODULE[\{H,\ i\},$
$\quad\quad i \ := \ n;$
$\quad\quad F \ := \ 1;$
$\quad\quad H \ := \ 1;$
$\quad\quad WHILE[i \ > \ 1,$
$\quad\quad\quad H \ := \ H \ + \ F;$
$\quad\quad\quad F \ := \ H \ - \ F;$
$\quad\quad\quad i \ := \ i \ - \ 1]]]$

Note: $FibExp[n]$ denotes the term: $F_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi}$ is its conjugate.
From the loop–body, we can set up the recurrence:

$$H_n = H_{n-1} + F_{n-1} + [n = 1], \ (n \in Z),$$
$$H_0 = 0$$
$$F_n = H_n - F_{n-1} \ (n \in Z),$$
$$F_0 = 0$$

where the value of $[n = 1]$ is 1 (i.e. H_1) when $n = 1$, and 0 when $n > 1$.

We seek a closed form for (F_n) and (H_n), using the technique of *generating functions* [33]. First, we obtain the harmonic forms ($n \in Z$):

$$\begin{aligned} H(z) &= \sum_n H_n z^n = \\ &= \sum_n H_{n-1} z^n + \sum_n F_{n-1} z^n + \sum_n [n = 1] z^n \\ &= zH(z) + zF(z) + z \\ F(z) &= \sum_n H_n z^n - \sum_n F_{n-1} z^n \\ &= H(z) - zF(z) \end{aligned}$$

Solving this system in the unknowns F and H, we obtain the generating functions:

$$F(z) = \frac{z}{1 - z - z^2}, \quad H(z) = \frac{z(1+z)}{1 - z - z^2}$$

and by the *rational expansion theorem* [15] their closed form:

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

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: $i_k = n - (k-1)$. Now we eliminate k from the three equations, obtaining:

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

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

2.4 Further Work

We have been able to automatically generate verification conditions (including invariants and termination terms) for several interesting programs, like: computation of the integer square/cubic root, sum of integers, square calculation, binary powering, etc [20]. These experiments show the power and also the limitations of the methods we have implemented.

One further development which we are now studying is the generation of invariants for nested WHILE loops. The main idea is to characterize the behavior of the WHILE loops using recurrence equations. Namely, we first generate the closed form for the critical variables of the innermost loop. These can be used in order to express the critical variables of the outer loop, and finally we try to obtain the invariant relations by elimination of the loop index variables.

Furthermore, we would like to integrate in our method an algorithm that generates, by combinatorial and algebraic methods, invariant properties for loops that contain also conditional statements (IF-THEN-ELSE). An efficient method for solving this problem is the application of Gröbner Bases [1], namely to generate the Gröbner bases of the obtained polynomial relations from several possible program-traces (this approach has also been investigated in [29,25,32]).

3 Functional Programs

While proving [partial] correctness of non-recursive procedural programs is quite well understood, for instance by using Hoare Logic [16,6], there are relatively few approaches to recursive procedures (see e.g. [27] Chap. 2).

We discuss here a practical approach, based on the fixpoint theory of programs and including implementation, for automatic generation of verification conditions for functional recursive programs. The implementation is part of the *Theorema* system, and complements the research performed in the *Theorema* group on verification and synthesis of functional algorithms based on logic principles [9,2,17].

We consider the correctness problem expressed as follows: *given* the program (by its source text) which computes the function F and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$, *generate* the verification conditions which are [minimally] sufficient for the program to satisfy the specification. This actually means that the function F satisfies the specification, that is: F terminates on any input x satisfying I_F, and, for each such input, the condition $O_F[x, F[x]]$ holds. One also calls this "total correctness" of the program.

The functional program of F can be interpreted as a set of predicate logic formulae, and the correctness of the program can be expressed as:

$$(\forall x : I_F[x])\ O_F[x, F[x]], \tag{1}$$

which we will call the *correctness formula* of F. Under the assumption of termination, for program correctness it is sufficient that the correctness formula is a logical consequence of the formulae corresponding to the definition of the function (and the specific theory which describes the properties of the domain[s] and the auxiliary functions involved). This approach was previously used by other authors and is also experimented with in the Theorema system [9]. However, the proof of (1) may be difficult, because the prover has to find the appropriate induction principle and has to find out how to use the properties of the auxiliary functions present in the program.

The method presented in this section generates several verification conditions, which are easier to prove. In particular, only the termination condition needs an inductive proof, and this termination condition is "reusable", because it basically expresses an induction principle which may be useful for several programs. This is important for automatic verification embedded in a practical verification system, because it leads to early detection of bugs (when proofs of simpler verification conditions fail).

Moreover, the verification conditions are provable in the frame of predicate logic, without using any theoretical model for program semantics or program execution, but only using the theories relevant to the predicates and functions present in the program text. This is again important for the automatic verification, because any additional theory present in the system will significantly increase the proving effort.

The study of recursive schemes starts its traditions in the seventies [12] and is still alive [30] where equivalence between different schemes have been proven. However, we are interested on particular schemes for deriving verification conditions which will then be used for verification of programs rather than schemes. The rules (conditions) for partial correctness are developed using Scott induction and the fixpoint theory of programs, however the verification conditions themselves do not refer to this theory, they only state facts about the predicates and functions present in the program text. In particular, the termination condition consists in a property of a certain simplified version of the original program.

We approach the correctness problem by splitting it into two parts: *partial correctness* (prove that the program satisfies the specification provided it terminates), and *termination* (prove that the program always terminates).

3.1 Simple Recursive Programs.

In this paper we study the recursive scheme which is the most used in practice and at the same time the most elementary one, namely we look at programs of the form:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]], \tag{2}$$

where Q is a predicate[1] and S, C, R are auxiliary functions ($S[x]$ is a "simple" function, $C[x, y]$ is a "combinator" function, and $R[x]$ is a "reduction" function). We assume that the functions S, C, and R satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_C[x, y]$, $O_C[x, y, z]$, $I_R[x]$, $O_R[x, y]$. Note that functions with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Note that the "programming language" used here contains only the construct **If–then–else** in addition to the language of first order predicate logic. One may also use some additional restrictions on the shape of the definitions of Q, S, C, and R (e. g. that they do not contain quantifiers) in order to make the program "easy" to execute. However, this depends on the complexity of the "interpreter" ("compiler") and does not influence the actual generation of the verification conditions. In general, the auxiliary functions may be already defined in the underlying theory, or by other programs (that includes logical terms).

Now we are ready to formulate the method and its soundness in the form of the following theorem.

Theorem 1 *Let S, C, and R be functions which satisfy their specifications. Then the program (2) satisfies the specification given by I_F and O_F if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \ \Rightarrow O_F[x, S[x]]) \tag{3}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \ \Rightarrow I_F[R[x]]) \tag{4}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow O_F[x, C[x, F[R[x]]]]) \tag{5}$$

$$(\forall x : I_F[x])\ (Q[x] \ \Rightarrow I_S[x]) \tag{6}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \ \Rightarrow I_R[x]) \tag{7}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \wedge O_F[R[x], F[R[x]]] \Rightarrow I_C[x, F[R[x]]]) \tag{8}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{9}$$

where:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } F[R[x]] \tag{10}$$

Proof. First we will see that (2) terminates. Indeed, from the assumption that S, C, and R are totally correct (with respect to I_S, I_C, and I_R) by (6), (7), and (8) we ensure the *termination* of the calls to the auxiliary functions S, C, and R.

The condition (9) expresses that a simplified version F' of the initial function F terminates. Moreover, F terminates if F' terminates.

The precise proof of the termination of F goes as follows: Take arbitrary but fixed x and assume $I_F[x]$. From (9), we obtain that $F'(x) = 0$. We first show that there must exist a number n such that after n steps of recursive calls, the predicate Q will be satisfied, that is

$$F'(x) = 0 \Rightarrow (\exists n \in \mathbb{N})(Q[R^n[x]]), \tag{11}$$

where $R^0[x] = x$ and $R^{n+1}[x] = R[R^n[x]]$. We prove this statement by contradiction, i.e. assume:

$$F'(x) = 0 \wedge (\forall n \in \mathbb{N})(\neg Q[R^n[x]]).$$

Henceforth, by $\downarrow$ we denote the predicate expressing termination (we want to prove $F[x] \downarrow$) and by Ω the nowhere defined function and by $\perp$ the nonterminating term – $\forall x \Omega[x] = \perp$.

Let $f_0, f_1, ...f_m...$ be the finite approximations of F' obtained as

$$f_0[x] = \Omega[x]$$

$$f_{m+1}[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } f_m[R[x]],$$

then the function computed by (10) is defined as

$$F' = \bigcup_m f_m$$

which is the least fixpoint of (10).

Since we have $F'(x) = 0$, there must exist a finite approximation f_m, such that $f_m[x] = 0$. If $m = 0$ then $f_0[x] = 0$ which contradicts the definition of $f_0 = \Omega$, hence $m > 0$.

From $(\forall n \in \mathbb{N})(\neg Q[R^n[x]])$ and in particular $\neg Q[x]$ by the definition of f_m we obtain $f_m[x] = f_{m-1}[R[x]]$. By repeating the same kind of reasoning m times, we obtain that $f_m[x] = f_0[R^m[x]]$ and by its definition $f_0[R^m[x]] =$

[1] In practice Q may also be implemented by a program, and it may also have an input condition, but we do not want to complicate the present discussion by including this aspect, which has a special flavor.

$\perp$ which contradicts $f_m[x] = 0$, and so, we have proven (11).

The proof of the termination of F will be completed by proving the following statement:

$$(\exists n \in \mathbb{N})(Q[R^n[x]] \Rightarrow F[x] \downarrow). \tag{12}$$

Assume $Q[R^n[x]$ for some n and, without loss of generality, assume $(\forall k < n)(\neg Q[R^k[x]])$.

– Case 1: $n = 0$. By the definition of F, $F[x] = S[x]$ and by (6) we obtain that $S[x] \downarrow$ and hence $F[x] \downarrow$.

– Case 2: $n > 0$. By unfolding the definition of F, we obtain $F[x] = C[x, F[R[x]]] = C[x, C[R[x, F[R^2[x]]]]]...$ where using (7) and (8) we ensure termination on each step. Finally, we obtain $F[x] = C[..., S[R^n[x]]]$ where the termination is ensured by (6). Using induction one may construct a detailed proof, however, we skip the details here.

Secondly, using Scott induction, we will show that (2) is partially correct:

$$(\forall x : I_F[x])(F[x] \downarrow \Rightarrow O_F[x, F[x]]). \tag{13}$$

As it is known [22,23], not every property is admissible and may be proven by Scott induction. However, properties which express partial correctness (13) are known to be admissible. A property ϕ is said to be partial correctness property if and only if there are predicates I and O, such that:

$$(\forall f)(\phi[f] \Leftrightarrow (\forall a)(f[a] \downarrow \wedge I[a] \Rightarrow O[a, f[a]])). \tag{14}$$

We now consider the following partial correctness property ϕ:

$$(\forall f)(\phi[f] \Leftrightarrow (\forall a)(f[a] \downarrow \wedge I_F[a] \Rightarrow O_F[a, f[a]])).$$

The first step in Scott induction is to show that ϕ holds for the *nowhere defined function* Ω. By the definition of ϕ we obtain

$$\phi[\Omega] \Leftrightarrow (\forall a)(\Omega[a] \downarrow \wedge I_F[a] \Rightarrow O_F[a, \Omega[a]])),$$

and so, $\phi[\Omega]$ holds, since $\Omega[a] \downarrow$ never holds.

In the second step of Scott induction we assume $\phi[f]$ for some f:

$$(\forall a)(f[a] \downarrow \wedge I_F[a] \Rightarrow O[a, f[a]]), \tag{15}$$

and show $\phi[f']$, where f' is obtained from f by the main program (2) as follows:

$$f' = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, f[R[x]]].$$

We need to show now that for an arbitrary a,

$$f'[a] \downarrow \wedge I_F[a] \Rightarrow O_F[a, f'[a]]$$

and so, assume $f'[a] \downarrow$ and $I_F[a]$.

– Case 1: $Q[a]$. By the definition of f' we obtain $f'[a] = S[a]$ and since $f'[a] \downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a] \downarrow$. Now using verification condition (3) we may conclude $O_F[a, S[a]]$ and so $O_F[a, f'[a]]$.

– Case 2: $\neg Q[a]$. By the definition of f' we obtain $f'[a] = C[a, f[R[a]]]$ and since $f'[a] \downarrow$, we conclude that all the others involved in this computation must also terminate, that is: $C[a, f[R[a]]] \downarrow$, $f[R[a]] \downarrow$, and $R[a] \downarrow$. From $I_F[a]$, by (4), we obtain $I_F[R[a]]$ and, knowing that $f[R[a]] \downarrow$, by the induction hypothesis (15) we obtain $O_F[R[a], f[R[a]]]$.

Concerning the verification condition (5), note that all the assumptions from the left part of the implication are at hand and thus we can conclude $O_F[a, f'[a]]$.

Now we conclude that the property ϕ holds for the least fixpoint of (2) and hence, ϕ holds for the function computed by (2), which completes the proof of the theorem.

One can easily extend this method to a more general one, by adding an appropriate treatment for programs defined by **Case** (**If–then–else** with several cases), as illustrated in the example below. The new soundness theorem is very similar – the conditions (3), (6), and (9) remain basically the same and for each "else" branch we generate conditions similar to (4), (5), (7), and (8). The proof will follow the same kind of steps.

3.2 Example and Discussion.

For illustrating the method we give here an example of a *binary powering* program annotated with its specification. Let the program be:

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

We consider this program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n.$$

The (automatically generated) verification conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \Rightarrow 1 = x^n) \tag{16}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \text{Even}[n] \Rightarrow n/2 \in \mathbb{N}) \tag{17}$$

$$(\forall x, n, m : n \in \mathbb{N}) \tag{18}$$

$$(n > 0 \wedge \text{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow m = x^n)$$

$$(\forall x, n : n \in \mathbb{N}) \tag{19}$$

$$(n > 0 \wedge \neg \mathrm{Even}[n] \ \Rightarrow (n-1)/2 \in \mathbb{N})$$

$$(\forall x, n, m : n \in \mathbb{N}) \tag{20}$$

$$(n > 0 \wedge \neg \mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2} \ \Rightarrow x * m = x^n)$$

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \ \Rightarrow \mathbb{T}) \tag{21}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \mathrm{Even}[n] \ \Rightarrow \mathrm{Even}[n]) \tag{22}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \neg \mathrm{Even}[n] \ \Rightarrow \mathrm{Odd}[n]) \tag{23}$$

$$(\forall x, n, m : n \in \mathbb{N}) \tag{24}$$

$$(n > 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \ \Rightarrow \mathbb{T})$$

$$(\forall x, n, m : n \in \mathbb{N}) \tag{25}$$

$$(n > 0 \wedge \neg \mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2} \ \Rightarrow \mathbb{T})$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = 0, \tag{26}$$

where

$$\begin{aligned} P'[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P'[x * x, n/2] \\ & \textbf{else } P'[x * x, (n-1)/2]. \end{aligned}$$

One sees that the formulae (21), (24) and (25) are trivial to prove – the truth constant $\mathbb{T}$ is at the right side of an implication. The formulae (16) to (20), (22) and (23) are easy consequences of the elementary theory of reals and naturals. The only formula which needs an induction proof (on the second argument n) is (26).

As already mentioned, the termination condition (9) (which is (26) in the example above) is reusable. We are collecting proven (or assumed to be obviously valid) "simplified versions" and we are putting them into a library for further reusability. The most basic element of this library is:

$$Prim[n] = \ \textbf{If } n = 0 \textbf{ then } 0 \textbf{ else } Prim[n-1],$$

where $(\forall n : n \in \mathbb{N})$, because it represents all the primitive recursive programs on one argument.

Currently we are working on extending the method in order to cover more general program schemes, such that they include multiple recursive calls, mutual recursive programs, etc. Another research direction is proving minimality of the verification conditions – that would lead to a "Completeness" theorem, which would then provide a better environment for debugging.

4 Conclusions and Further Work

Using several complementary approaches to the generation of verification conditions, we obtain more insight into the issue of practical algorithm verification in the context of automatic reasoning. Moreover, we obtain interesting proof problems for the automatic provers of the *Theorema* system, which allow us to detect the possible weaknesses of the current provers and to design and implement new proving methods.

Further work includes the extension of these methods to more complex programs, as well as a more systematic comparison and cross–fertilisation of the different approaches to the generation of verification conditions both for functional and for imperative programs.

References

1. Buchberger, B.: *Introduction to Gröbner Bases.* In: Gröbner Bases and Applications, London Mathematical Society Lecture Notes Series 251, Cambridge University Press (1998). pp. 3-31
2. Buchberger, B.: Verified algorithm development by lazy thinking. In: Proc. of IMS 2003. Imperial College, London (2003)
3. Buchberger, B., Craciun, A.: Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In: Electronic Notes in Theoretical Computer Science, vol. 93, pp. 24–59 (2004). Proc. of MKM 2003
4. 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 (2006). To appear
5. Buchberger, B., Dupre, C., Jebelean, T., Kriftner, F., Nakagawa, K., Vasaru, D., Windsteiger, W.: The Theorema Project: A Progress Report. In: Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning (2000)
6. Buchberger, B., Lichtenberger, F.: *Mathematics for Computer Science I - The Method of Mathematics (in German)*, 2nd edn. Springer (1981)
7. Colon, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear Invariant Generation Using Non-Linear Constraint Solving. In: Proc. of CAV 2003, *LNCS*, vol. 2725, pp. 420–432 (2003)
8. 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). Tucson, Arizona
9. Craciun, A., Buchberger, B.: Functional program verification with theorema. In: Proc. of CAVIS-03. Institute e-Austria Timisoara (`www.ieat.ro`), Romania (2003)
10. Dijkstra, E.W.: *A Discipline of Programming.* Prentince-Hall (1976)
11. Futschek, G.: *Programmentwicklung und Verifikation.* Springer (1989)
12. Garland, S., Luckham, D.: Program schemes, recursion schemes, and formal languages. Journal of Computer and Systems Sciences **7/2**, 119–160 (1973)
13. Goldberg, B., Zuck, L., Barrett, C.: Into the loops. In: Proc. of COCV 2004 (2004)
14. Gosper, R.W.: *Decision procedures for indefinite hypergeometric summation.* Journal of Symbolic Computation **75**, 40–42 (1978)

15. Graham, R.L., Knuth, D.E., Patashnik, O.: *Concrete Mathematics, 2nd ed.* Addison-Wesley Publishing Company (1989). pp. 306-330
16. Hoare, C.A.R.: *An axiomatic basis for computer programming.* Comm. ACM **12** (1969)
17. Jebelean, T., Kovacs, L., Popov, N.: Verification of imperative programs in Theorema. In: Proc. of SEEFM03 (2003). Thessaloniki, Greece
18. Kirchner, M.: Program verification with the mathematical software system Theorema. Tech. Rep. 99-16, RISC-Linz, Austria (1999). PhD Thesis
19. Knuth, D.E.: *The Art of Computer Programming, volume 2 / Seminumerical Algorithms*, 3rd edn. Addison-Wesley (1998)
20. Kovacs, L.: Using combinatorial and algebraic techniques for automatic generation of loop invariants. Tech. Rep. 05-16, RISC-Linz, Austria (2005)
21. Kovacs, L., Jebelean, T.: Practical Aspects of Imperative Program Verification in Theorema. In: Proc. of Synasc 2003, pp. 317–320. Timisoara, Romania (2003)
22. Loeckx, J., Sieber, K.: The Foundations of Program Verification, 2nd edn. Teubner (1987)
23. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill Inc. (1974)
24. Müller-Olm, M., Petter, M., Seidl, H.: Interprocedurally Analyzing Polynomial Identities. In: Proc. of STACS 2006 (2006). Marseille, France
25. Müller-Olm, M., Seidl, H.: *Polynomial Constants are Decidable.* In: Proc. of SAS 2002, vol.2477 of LNCS (2002). pp. 4-19
26. 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)
27. Paull, M.C.: Algorithm Design. A recursion transformation framework. Wiley (1987)
28. Popov, N., Jebelean, T.: A practical approach to verification of recursive programs in theorema. In: Proc. of SYNASC 2003 (2003). Timisoara, Romania
29. Rodriguez-Carbonell, E., Kapur, D.: *Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations.* In: Proc. of ISSAC 04 (2004). Santander, Spain
30. Sabelfeld, V.: The tree equivalence of linear recursion schemes. Theoretical Computer Science **238/1-2**, 1–29 (2000)
31. Salvy, B., Zimmermann, P.: Gfun: A package for the manipulation of generating and holonomic functions in one variable. ACM Trans. Math. Software **20**, 163–177 (1994)
32. Sankaranaryanan, S., Henry, B.S., Manna, Z.: *Non-Linear Loop Invariant Generation using Gröbner Bases.* In: Proc. of POPL 2004 (2004). Venice, Italy
33. Stanley, R.P.: Differentiably finite power series. European Journal of Combinatorics **1**, 175–188 (1980)
34. Wolfram, S.: The Mathematica Book. Version 5.0. Wolfram Media (2003)