Available online at www.sciencedirect.com
ScienceDirect
Mathematics and Computers in Simulation 79 (2009) 2302–2309
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Using Computer Algebra techniques for the specification, verification and synthesis of recursive programs☆

Nikolaj Popov, Tudor Jebelean *

Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria

Available online 7 December 2008

Abstract

We describe an innovative method for proving total correctness of tail recursive programs having a specific structure, namely programs in which an auxiliary tail recursive function is driven by a main nonrecursive function, and only the specification of the main function is provided. The specification of the auxiliary function is obtained almost fully automatically by solving coupled linear recursive sequences with constant coefficients. The process is carried out by means of CA (Computer Algebra) and AC (Algorithmic Combinatorics) and is implemented in the *Theorema* system (using *Mathematica*). We demonstrate this method on an example involving polynomial expressions. Furthermore, we develop a method for synthesis of recursive programs for computing polynomial expressions of a fixed degree by means of "cheap" operations, e.g., additions, subtractions and multiplications. For a given polynomial expression, we define its recursive program in a schemewise manner. The correctness of the synthesized programs follows from the general correctness of the synthesis method, which is proven once for all, using the verification method presented in the first part of this paper.

Keywords: Specification and verification; Program synthesis; Computer Algebra

1. Introduction

Formal verification, in general, is the act of proving the correctness of a program with respect to a certain formal specification. There are roughly two main approaches: *Model checking* [4], which consists of an exploration of the mathematical model—that is only possible for finite models; *Program Verification*– where our contribution falls — consists of using logical reasoning about the program, usually with the help of a theorem prover [10].

The problem of verifying programs is usually split into two subproblems: generate verification conditions which are sufficient for the program to be correct and prove the verification conditions, within the theory of the domain for which the program is defined. In this paper, we address both of these subproblems.

Program synthesis is the act of (automatically) developing correct programs from formal specifications [6]. The verification of the synthesized programs is done usually at the meta-level, that is, the synthesis method is accomplished by a proof of its correctness.

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

* Corresponding author.
E-mail addresses: popov@risc.uni-linz.ac.at (N. Popov), jebelean@risc.uni-linz.ac.at (T. Jebelean).

0378-4754/$36.00 © 2008 IMACS. Published by Elsevier B.V. All rights reserved.
doi:10.1016/j.matcom.2008.11.017

Our work is performed in the frame of the *Theorema* system, a mathematical computer assistant which aims at supporting all the phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. The system includes a collection of general as well as specific provers for various interesting domains (e.g., integers, sets, reals, tuples, etc.). An overview of *Theorema* could be found in [3].

2. Verification of tail recursive programs

In this section, we present an innovative method for proving total correctness of tail recursive programs of a specific configuration, namely, programs in which a tail recursive subordinate function, henceforth A, is driven by a main nonrecursive function M, for which alone a specification is provided. The specification of the auxiliary function – and this is the main contribution of this section – is obtained, almost fully automatically by solving coupled linear recursive sequences with constant coefficients (see, e.g., [8], Chapter 7). The solving step is carried out by means of an algorithm available in *Mathematica* [15].

It is impossible to find automatically, in general, verification conditions for an arbitrary tail recursive function without knowing its specification. However, in many particular cases this is, nevertheless, possible.

The approaches of the classical books in program verification (see, e.g., [11,10]) do not address the problem of unspecified functions. However, in practice, one does not always have specifications, and our work contributes towards addressing this situation. While other program verification tools are either not dealing with recursion over the reals or they need explicitly to use the specification of the program and a termination term (see [13,9,2]), our method successfully resolves both these challenges. Furthermore, we are able to prove, fully automatically, termination for a certain class of problems (see [12]).

We pose the problem in the following way: Given a tail recursive function (program), defined by a main function and an auxiliary function and given the specification of the main function; prove total correctness of the main function with respect to the specification. The method we are presenting consists briefly of the following steps: find what the auxiliary function does, that is, find its closed form; prove termination of the auxiliary function; prove correctness of the main function.

Let M be the main program defined as:

$$M[\bar{a}] = Second[A[R[\bar{a}], \bar{c}]] \tag{1}$$

where the auxiliary program A is defined as:

$$A[x, y, \ldots, z] = \text{If } Q[x, y, \ldots, z] \text{ then } \langle x, y, \ldots, z\rangle \text{ else } A[a_1x + b_1y + \ldots + c_1z + d_1, a_2x + b_2y + \ldots + c_2z + d_2, \ldots]. \tag{2}$$

Here, by $\langle x, y, \ldots, z\rangle$ we mean the tuple of x, y,..., z, by *Second* the corresponding function on tuples, Q is a predicate like $<$, $=$, $>$, $\leq$, $\geq$, $\neq$ and $\bar{c}$, a_1, b_1, c_1, d_1, etc. are constants and $\bar{a}$, $\bar{c}$ denote tuples. Given, additionally, the specification of (1) by a precondition I_M (a predicate on the input variables, expressing their possible values) and a postcondition O_M (a predicate on the input variables and the output expressing the relation between them, i.e., what the output should be), we want to prove the total correctness of M with respect to the specification, that is:

$$\forall \bar{a}(I_M[\bar{a}] \Rightarrow O_M[\bar{a}, M[\bar{a}]]), \tag{3}$$

and the program terminates on any argument satisfying I_M.

For illustrating the method, we follow an example during the whole presentation. We have chosen a typical example of a tail recursive function, which is taken from [7] and has been used for demonstrating other program verification tools. Let $CubR$ be the program (function) (4) for computing an approximative cubic root from a given real number. In other words, given a real number a, $CubR[a]$ computes the integer number which is nearest to $\sqrt[3]{a}$.

$$CubR[a] = Third\left[aux\left[a, \frac{13}{4}, 1\right]\right], \tag{4}$$

where:

$$aux[x, s, r] = \text{If } x \leq s \text{ then } \langle x, s, r\rangle \text{ else } aux[x - s, s + 6r + 3, r + 1]. \tag{5}$$

The specification of (4) is as follows:

$$I_{CubR}[a] \Leftrightarrow (a \in \mathbb{R} \bigwedge a \geq 0);$$

$$O_{CubR}[a, CubR[a]] \Leftrightarrow CubR[a] \in \mathbb{N} \bigwedge \left(CubR[a] - \frac{1}{2}\right)^3 < a \bigwedge \left(CubR[a] + \frac{1}{2}\right)^3 > a. \tag{6}$$

The method proceeds as follows.

2.1. Step one: obtain the closed form of the auxiliary function and prove its termination

Firstly, we consider the special case, namely when no recursion has been entered. The closed form of aux is expressed by:

$$\forall a \left(a \leq \frac{13}{4} \Rightarrow aux\left[a, \frac{13}{4}, 1\right] = \left\langle a, \frac{13}{4}, 1\right\rangle\right)$$

Secondly, we consider the case when the recursive loop is entered. The method for obtaining the closed form is based on solving coupled linear recursive sequences with constant coefficients (see, e.g., [8], Chapter 7). The later one is implemented in *Mathematica* [15]. In more detail, this is done by solving first the recurrence:

$$\begin{cases} x_n = a_1 x_{n-1} + b_1 y_{n-1} + \ldots + c_1 z_{n-1} + d_1 \\ y_n = a_2 x_{n-1} + b_2 y_{n-1} + \ldots + c_2 z_{n-1} + d_2 \\ \ldots \\ z_n = a_k x_{n-1} + b_k y_{n-1} + \cdots + c_k z_{n-1} + d_k \end{cases} \tag{7}$$

and obtaining a closed form of $x_n, y_n, \ldots, z_n$ depending on $x_0, y_0, \ldots, z_0$, and the constants $a_1, b_1, \ldots$ only. In the system we want to solve, $x_0, y_0, \ldots, z_0$ correspond to the initial values of $x, y, \ldots, z$, and for any i, $x_i, y_i, \ldots z_i$ correspond to the values of $x, y, \ldots, z$ after i recursive steps.

For the purpose of our example we describe this in even more detail. Consider what (5) does for one step of execution of the recursive loop. It takes $\langle x_0, s_0, r_0\rangle$ and it returns $\langle x_1, s_1, r_1\rangle$, where $x_1 = x_0 - s_0$, $s_1 = s_0 + 6r_0 + 3$ and $r_1 = r_0 + 1$. Here we assume that $\neg(x \leq s)$, otherwise, we would have said: zero steps of execution and thus go to the special case. Going further, we express this transformation in a matrix form:

$$\begin{pmatrix} x_1 \\ s_1 \\ r_1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_0 & -s_0 & & \\ & s_0 & +6r_0 & +3 \\ & & r_0 & +1 \\ & & & 1 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 6 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ s_0 \\ r_0 \\ 1 \end{pmatrix} \tag{8}$$

and we say that A_1 expresses one-step-execution:

$$A_1 = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 6 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9}$$

Now we express what (5) does for n steps of execution of the recursive loop:

$$(A_1)^n = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 6 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}^n == \begin{pmatrix} 1 & -n & -3(n-1)n & -\dfrac{n-3n^2+2n^3}{2} \\ 0 & 1 & 6n & 3n^2 \\ 0 & 0 & 1 & n \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{10}$$

Hence for n steps, $\langle x_0, s_0, r_0\rangle$ is transformed into $\langle x_n, s_n, r_n\rangle$, where:

$$\begin{aligned} x_n &= x_0 - ns_0 - 3(n-1)nr_0 - \frac{n-3n^2+2n^3}{2} \\ s_n &= s_0 - 6nr_0 + 3n^2 \\ r_n &= r_0 + n. \end{aligned}$$

From here, by substituting x_0, s_0 and r_0 by a, $13/4$ and 1 respectively, we obtain:

$$\begin{aligned} x_n &= a - \frac{13n}{4} - 3(n-1)n - \frac{n-3n^2+2n^3}{2} \\ s_n &= \frac{13n}{4} - 6n + 3n^2 \\ r_n &= 1 + n, \end{aligned}$$

which is a closed form of the recursive expression depending on the number of recursive steps n.

Now, by finding restrictions for termination of the recursion we will eliminate the number of steps n. For that we formulate the exit condition, when $Q[x_n, y_n, \ldots, z_n]$ at step n and the non-exit condition, when we have respectively $\neg Q[x_{n-1}, y_{n-1}, \ldots, z_{n-1}]$ at step $n-1$.

The exit and the non-exit conditions both are algebraic expressions (typically inequalities) involving x_0, y_0, …, z_0, n and the constants a_1, b_1, … only.

We now form a system of the two algebraic expressions and we solve it with respect to n and obtain a solution of the form:

$$n \in Interval_1 \cup Interval_2 \cup \ldots$$

In the example, the exit and the non-exit conditions are respectively $x_n \leq s_n$ and $x_{n-1} > s_{n-1}$, which after instantiation and simplification by using the Cylindrical Algebraic Decomposition (CAD) (see, e.g., [5]) method are transformed to:

$$a - \frac{1}{4}n(3 + 6n + 4n^2) \leq \frac{13}{4} + 6n + 3n^2, \tag{11}$$

$$\frac{1}{4}(1 + 4a - 3n + 6n^2 - 4n^3) > \frac{1}{4} + 3n^2, \tag{12}$$

where, we assume that $n > 0$; the other case was considered earlier.

After solving the system of equations (11), (12) and $n > 0$ with respect to n, again with the help of CAD, we obtain:

$$\frac{13}{4} < a \leq \frac{31}{2} \bigwedge 1 \leq n < Root[-4a + 3\#1 + 6\#1^2 + 4\#1^3, 1] \tag{13}$$

$$\frac{31}{2} < a \bigwedge Root[13 - 4a + 27\#1 + 18\#1^2 + 4\#1^3, 1] \leq n$$

$$\frac{31}{2} < Root[-4a + 3\#1 + 6\#1^2 + 4\#1^3, 1], \tag{14}$$

where the *Mathematica* [15] function $Root[f, k]$ represents the k th root of the polynomial equation $f[x] = 0$—in this case, the first (smallest) root of the polynomial $-4a + 3\#1 + 6\#1^2 + 4\#1^3 = 0$ in the variable #1 is chosen.

Now we prove existence of n, defined as such, which is equivalent to proving termination of A. Since we need the smallest n, from (13) we obtain $n = 1$, when $13/4 < a \leq 31/2$. Looking at (14) we may see the general phenomenon that the two polynomials are "shifted" by 1. If we rename them, e.g., $p[y] = 4y^3 + 18y^2 + 27y - 4a + 13$ and $q[y] = 4y^3 + 6y^2 + 3y - 4a$, we notice that $p[y-1] = q[y]$, which is due to the fact that we actually formed them on the steps n and $n-1$ of the recursion. Hence, there exists exactly one integer n, obeying (14). The existence of n, in fact, proves the termination of the auxiliary function A. Now, we are ready to express n as a function of $x_0, y_0, \ldots, z_0$:

$$n[x_0, y_0, \ldots z_0] = \begin{cases} 0 & \Leftarrow Q[x_0, y_0, \ldots z_0] \\ \mu_{n\in\mathbb{N}}\,(n \in Interval_1 \cup \cdots \cup \ldots) & \Leftarrow \text{otherwise} \end{cases} \tag{15}$$

where $\mu_{n\in\mathbb{N}}$ expresses "the first n" quantifier for $n \in \mathbb{N}$. In the example:

$$n[a] = \begin{cases} 0 & \Leftarrow a \leq \dfrac{13}{4} \\ 1 & \Leftarrow \dfrac{13}{4} < a \leq \dfrac{31}{2} \\ \mu_{n\in\mathbb{N}} \begin{pmatrix} 4n^3 + 18n^2 + 27n - 4a + 13 \geq 0 \\ \bigwedge \\ 4n^3 + 6n^2 + 3n - 4a < 0 \end{pmatrix} & \Leftarrow \text{otherwise} \end{cases} \tag{16}$$

While obtaining the closed form of the auxiliary function A, as a side effect we got a proof of its termination, proving the existence of n.

2.2. Step two: verification of the main function

After having obtained the closed form of the auxiliary function, the verification of the main function M consists of an appropriate substitution in the specification of M by A. In the example we have $CubR[a] = Third[aux[a, 13/4, 1]]$ and as obtained earlier $aux[a, 13/4, 1] = \langle x_n, s_n, n+1\rangle$, we conclude that $CubR[a] = n + 1$, where n is defined as $n[a]$ in (16). Verifying the main function reduces to proving (6), that is:

Assume:

$$n \in \mathbb{N} \bigwedge 4n^3 + 18n^2 + 27n - 4a + 13 \geq 0 \bigwedge 4n^3 + 6n^2 + 3n - 4a < 0 \tag{17}$$

Prove:

$$1 + n \in \mathbb{N} \tag{18}$$

$$\left(n + \frac{1}{2}\right)^3 < a \tag{19}$$

$$\left(n + \frac{3}{2}\right)^3 > a \tag{20}$$

Under the assumption that $n \in \mathbb{N}$ proving (18) is trivial. More interesting are the proofs of the other two conjuncts (19) and (20). The proving method we use now is known as "Proof by refutation". For that we take the negation of (19) and we put it to the assumption list (17) and try to solve the so formed system by using again CAD. Similarly, we do for (20). If the formed systems of inequations have no solutions, then the proof is successful. If any of the systems has a solution, that solution becomes a counter example of the proof. Indeed, when solving the systems, we found the following counter example: $n = 2$ and $a = 15.6$. Thus, *the verification fails*! Is the program really incorrect? $CubR[15.6] = 3$ versus $\sqrt[3]{15.6} = 2.49867$.

On purpose we used this example which fails, because we want to demonstrate the usefulness of our method in such situations also. Indeed, most texts about program verification present correct programs. However, in practical situations, it is the failure which occurs more often, until the program and the specifications are completely debugged. It is therefore very important for practical applications that the verification methods also give a meaningful answer in these situations, and this answer constitutes a help for debugging. This is also consistent with the philosophy of the

provers implemented in the *Theorema* system: these provers produce a detailed proof also in cases of failure, and since the proofs are presented in natural (human readable) style, the user can find the cause of the failure and correct the situation.

The only thing we could do is to change the program or the specification. For instance, we may provide a new specification for the main function—now the input argument a may only be a natural number:

$$I_{CubR}[a] \Leftrightarrow (a \in \mathbb{N} \bigwedge a \geq 0);$$

$$O_{CubR}[a, CubR[a]] \Leftrightarrow CubR[a] \in \mathbb{N} \bigwedge \left(CubR[a] - \frac{1}{2}\right)^3 < a \bigwedge \left(CubR[a] + \frac{1}{2}\right)^3 > a,$$

and by repeating the verification process we see that this new specification is fulfilled.

Moreover, the method presented here gives the *strongest* (exact) specification of the auxiliary program. For proving (verifying) any other (human) provided specification, it suffices to prove the implication *Strongest Specification* $\Rightarrow$ *Provided Specification*.

3. Synthesis of tail recursive programs for computing polynomial expressions

In this sections, we follow the natural escalation from program verification to program synthesis. Additional inspiration comes from the fact that many of the programs we want to verify are not correct, as it was shown in the example from the previous section.

We now describe a method for synthesis of recursive programs for computing polynomial expressions of third degree by means of "cheap" operations, e.g., additions, subtractions and multiplications. Even though the topic of "Fast Polynomial Evaluation" is well studied in the literature (see, e.g., [1,14]), the method presented here is very specific, however, very efficient.

We pose the problem in the following way: Given a real polynomial on r:

$$poly[r] = r^3 + cr^2 + br + a, \tag{21}$$

given an input parameter $inp (inp \in \mathbb{R})$; construct a program P using only cheap operations and recursion, such that,

$$P[inp] = \langle poly[n], n \rangle, \tag{22}$$

where n is the first integer, such that:

$$\begin{aligned} &poly[n] \leq inp < poly[n-1] \Leftarrow a > inp(poly[0] = a) \\ &n = 0 \Leftarrow a \leq inp(poly[0] = a) \end{aligned} \tag{23}$$

In other words, for a given polynomial $poly$, we need a program P, such that for any real input inp, $P[inp]$ computes the integer n for which $poly[n]$ approximates inp, and the approximation is computed as well.

The program listed below does the desired computation. In fact, P is actually a program scheme, because for any concrete polynomial coefficients c, b and a, there is a concrete program.

$$P[inp] = aux[c - b + a - 1 - inp, c - b - 1, 0], \tag{24}$$

where:

$$aux[x, y, m] = \text{If } x \leq y \text{ then } \langle x, y, m \rangle \text{ else } aux[x - y, y - 2c - 6m, m + 1]. \tag{25}$$

Specification. For the precondition we have one very natural, however, very important restriction on the input inp—the system

$$\left| \begin{array}{l} r^3 + cr^2 + br + a - inp \leq 0 \\ (r-1)^3 + c(r-1)^2 + b(r-1) + a - inp > 0 \end{array} \right. \tag{26}$$

must have a real nonnegative solution. If it does not have one, how can we find a natural n, obeying (23). Conversely, if (26) does have a nonnegative real solution then it has a natural one as well, because the two polynomials forming (26)

are "shifted" by 1, as discussed in the previous section. Thus, as a precondition of P we take a predicate I_P, such that

$$\forall inp \in \mathbb{R}(I_P[inp] \Leftrightarrow \exists r \in \mathbb{R}(r \geq 0) \bigwedge (r^3 + cr^2 + br + a - inp \leq 0) \bigwedge ((r-1)^3 + c(r-1)^2 + b(r-1) + a - inp > 0)). \tag{27}$$

For the postcondition we describe what the program must compute. In order to simplify the presentation, by x, y and m we denote respectively the first, the second and the third element of the output of P —it is defined to give a triple $\langle x, y, m\rangle$. In fact, $x - y$ accumulates the $poly[n]$, x accumulates $poly[n-1]$, and m accumulates the n itself.

$$\begin{aligned}&\forall inp, x, y, m \in \mathbb{R}(O_P[inp, \langle x, y, m\rangle]) \\ &\Leftrightarrow (m \in \mathbb{N}) \bigwedge (n > 0) \bigwedge (x - y \leq 0) \bigwedge (x > 0) \bigwedge (m^3 + cm^2 + bm + a - inp = x - y) \bigwedge ((m-1)^3 \\ &+ c(m-1)^2 + b(m-1) + a - inp = x) \bigvee (m = 0) \bigwedge (a \leq inp) \bigwedge (m^3 + cm^2 + bm + a - inp = x - y)\end{aligned} \tag{28}$$

Verification. The verification of P with respect to the specification ((27) and 28) is straightforward using the verification method introduced in the previous section.

As an example, we consider the synthesis of a program having the specification (6) of the example from the previous section.

$$I_{CubR'}[a] \Leftrightarrow (a \in \mathbb{R} \bigwedge a \geq 0)$$

$$O_{CubR'}[a, \langle x, y, m\rangle] \Leftrightarrow m \in \mathbb{N} \bigwedge \left(m - \frac{1}{2}\right)^3 < a \bigwedge \left(m + \frac{1}{2}\right)^3 > a \tag{29}$$

The synthesized program $CubR'$ is:

$$CubR'[a] = Third\left[aux\left[a + \frac{1}{8}, \frac{1}{4}, 0\right]\right], \tag{30}$$

where:

$$aux[x, y, m] = \text{If } x \leq y \text{ then } \langle x, y, m\rangle \text{ else } aux[x - y, y + 6m + 9, m + 1]. \tag{31}$$

Will this program meet the specification? Yes. Moreover, this time we do not need to verify; it is correct by the correctness of the synthesis method. In particular, note that, while $CubR[15.6] = 3$, the new program $CubR'[15.6] = 2$, which is correct since $\sqrt[3]{15.6} \approx 2.49$.

4. Conclusions

By using special (and quite powerful) methods from Computer Algebra, we are able to effectively verify relatively nontrivial programs involving polynomial expressions. In fact, the most difficult part of this verification is the automatic composition of the unknown specification of the auxiliary function. It is also notable that in case of failure to prove the correctness, the counterexample (that is, the "bug") can be found also automatically by using Cylindrical Algebraic Decomposition.

Moreover, we show in this paper that in fact it is even easier, by using powerful algebraic techniques from combinatorics, to actually synthesize programs of this type, instead of only verifying them. This synthesis method is in fact a contribution to the solution of a popular problem in Computer Algebra: fast evaluation of polynomials.

Further work based on the research presented here includes extension of the verification method to more complex programs, as well as the development of synthesis methods for higher degree polynomials.

References

[1] A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, 1st edition, Addison-Wesley, 1974.
[2] Y. Bertot, P. Casteran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Springer Verlag, 2004.
[3] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger, Theorema: towards computer-aided mathematical theory exploration, Journal of Applied Logic (2006) 470–504.
[4] E. Clarke, O. Grumberg, D. Peled, Model Checking, The MIT Press, 1999.
[5] G. Collins, Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition, Lecture Notes in Computer Science 33 (1975) 134–183.
[6] Pierre Flener, Achievements and prospects of program synthesis, in: Lecture Notes in Computer Science, vol. 2407, Computational Logic: Logic Programming and Beyond, 2002, pp. 310–346.
[7] P. Freire, Creations, http://www.pedrofreire.com, 2002.
[8] R.L. Graham, D.E. Knuth, O. Patashnik, Concrete Mathematics, 2nd edition, Addison-Wesley Publishing Company, 1989, pp. 306–330.
[9] M. Kaufmann, P. Manolios, J. Moore, Computer-Aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.
[10] J. Loeckx, K. Sieber, The Foundations of Program Verification, 2nd edition, Teubner, 1987.
[11] Z. Manna, Mathematical Theory of Computation, McGraw-Hill Inc., 1974.
[12] N. Popov, T. Jebelean, A practical approach to proving termination of recursive programs in Theorema, in: M. Codish, A. Middeldorp (Eds.), Proceedings of 7th International Workshop on Termination, Aachen, Germany, June 2004, pp. 43–46.
[13] PVS Group, PVS Specification and Verification System, 2004, http://pvs.csl.sri.com.
[14] John H. Reif, Approximate complex polynomial evaluation in near constant work per point, in: Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing (STOC), 1997, pp. 30–39.
[15] S. Wolfram, The Mathematica Book, 3rd edition, Wolfram Media/Cambridge University Press, 1996.