Functional Program Verification in Theorema. Soundness and Completeness

Nikolaj Popov, Tudor Jebelean*

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{popov,jebelean}@risc.uni-linz.ac.at

Abstract. We present a method for verifying recursive functional programs. We define a Verification Condition Generator (VCG) which covers the most frequent types of recursive programs. These programs may operate on arbitrary domains. Soundness and Completeness of the VCG are proven on the meta level, and this provides a warranty that any system based on our results will be sound.

1 Introduction

We present an experimental prototype environment for defining and verifying recursive functional programs, which is part of the *Theorema* system. In contrast to classical books on program verification [6],[4],[10] which expose methods for verifying correct programs, we put special emphasize on verifying incorrect programs. The user may easily interact with the system in order to correct the program definition or the specification.

There are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [12],[1],[2]), and this is where our contribution falls into. As a distinctive feature of our prototype is the hint on "what is wrong" in case of a verification failure.

This work is performed in the frame of the *Theorema* system [3], a mathematical computer assistant which aims at supporting all 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.). More details about *Theorema* could be found at `www.theorema.org`.

* This research was partly supported by BMBWK (Austrian Ministry of Education, Science, and Culture) and Upper Austrian Government Project "Technologietransferaktivitäten".

2 Programming, Specification and Verification

As usual, correctness is transformed into a set of first-order predicate logic formulae – verification conditions. As a distinctive feature of our method, these formulae are not only sufficient, but also necessary for the correctness [7]. We demonstrate our method on a relatively simple example, however, it show how correctness may be proven fully automatically. In fact, even if a small part of the specification is missing – in the literature this is often a case – the correctness cannot be proven. Furthermore, a relevant counterexample may be constructed automatically.

We consider the total correctness problem expressed as follows: *given* the program which computes the function F in a domain D 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 $VC_1, \dots , VC_n$ which are sufficient for the program to satisfy the specification. The function F satisfies the specification, if: for any input x satisfying I_F, F terminates on x, (we write $F[x] \downarrow$) and the condition $O_F[x, F[x]]$ holds:

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

A Verification Condition Generator (VCG) is a device—normally implemented by a program—which takes a program, actually its source code, and the specification, and produces verification conditions. These verification conditions do not contain any part of the program text, and are expressed in a different language, namely they are logical formulae.

Any VCG should come together with its *Soundness* statement, that is: for a given program F, defined on a domain D, with a specification I_F and O_F if the verification conditions $VC_1, \dots , VC_n$ hold in the theory $Th[D]$ of the domain D, then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct. A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on “what is wrong”. Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

3 Coherence and Verification Conditions

Before performing the “real” verification, we first make sure that our programs are coherent. It is not that programs which are not coherent are necessarily not correct, however, in order to construct a system of programs preserving modularity, we need to use only coherent programs.

3.1 Coherent Programs

In this subsection we state the principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [8]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

– start from basic (trustful) functions e.g. addition, multiplication, etc.;

– define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we “forbid” using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

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

The specification of P is:
The domain $\mathbb{D} = \mathbb{R}^2$, precondition $I_P[x,n] \iff n \in \mathbb{N}$ and a postcondition $O_P[x,n,P[x,n]] \iff P[x,n] = x^n$.

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

Furthermore, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs – we call this:

Appropriate values for the auxiliary functions. The following example will give an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

with the specification of F (I_F and O_F) and specifications of the auxiliary functions H (I_H and O_H), G (I_G and O_G). The two verification conditions, ensuring that the calls to the auxiliary functions have appropriate values are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

3.2 Recursive Programs and Generation of Verification Conditions

As is well-known, there is no universal VCG. Thus, in our research, we concentrate on constructing a VCG which is appropriate only for a certain kind of recursive programs – those which are defined by multiple choice *if-then-else* with zero, one, or more recursive calls on each branch (but without nested recursion). They are defined as those F:

$$F[x] = \ \textbf{If}\ Q_0[x]\ \textbf{then}\ S[x] \tag{2}$$

$$\begin{aligned} &\textbf{elseif}\ Q_1[x]\ \textbf{then}\ C_1[x, F[R_1[x]]]\\ &\textbf{elseif}\ Q_2[x]\ \textbf{then}\ C_2[x, F[R_2[x]]]\\ &\dots\\ &\textbf{else}\ Q_n[x]\ \textbf{then}\ C_n[x, F[R_n[x]]]. \end{aligned}$$

where Q_i are predicates and S, C_i, R_i are auxiliary functions ($S[x]$ is a “simple” function (the bottom of the recursion), $C_i[x, y]$ are “combinator” functions, and $R_i[x]$ are “reduction” functions). We assume that the functions S, C_i, and R_i satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$. Additionally, assume that the Q_i predicates are non-contradictory, that is $Q_{i+1} \Rightarrow \neg Q_i$ and $Q_n = \neg Q_0 \wedge \dots \wedge \neg Q_{n-1}$, which we do only in order to simplify the presentation.

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.

Considering Coherent Recursive programs, we give here the appropriate definition:

Let S, C_i, and R_i be functions which satisfy their specifications. Then the program (2) is coherent if the following conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies I_S[x]) \tag{3}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_F[R_1[x]]) \tag{4}$$

$$\dots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_F[R_n[x]]) \tag{5}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_{R_1}[x]) \tag{6}$$

$$\dots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_{R_n}[x]) \tag{7}$$

$$(\forall x : I_F[x])(Q_1[x] \land O_F[R_1[x], F[R_1[x]]] \implies I_{C_1}[x, F[R_1[x]]]) \tag{8}$$

$$\dots$$

$$(\forall x : I_F[x])(Q_n[x] \land O_F[R_n[x], F[R_n[x]]] \implies I_{C_n}[x, F[R_n[x]]]). \tag{9}$$

It is not that a program which is not coherent is necessarily not correct. However, non-coherent programs are somehow inconsistent, namely proving their correctness would involve knowledge about their auxiliary functions which is out of the official scope. Thus, if we allow them in our system of verified programs, the modularity would be lost.

After performing the coherence check, we go to the verification. The upcoming theorem gives the necessary and sufficient conditions for a program to be correct. These conditions are taken as the *Verification Conditions.*

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

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{10}$$

$$(\forall x : I_F[x])(Q_1[x] \land O_F[R_1[x], F[R_1[x]]] \implies O_F[x, C_1[x, F[R_1[x]]]]) \tag{11}$$

$$\dots$$

$$(\forall x : I_F[x])(Q_n[x] \land O_F[R_n[x], F[R_n[x]]] \implies O_F[x, C_n[x, F[R_n[x]]]]) \tag{12}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{13}$$

where F' is defined as:

$$F'[x] = \textbf{ If } Q_0[x] \textbf{ then } 0 \tag{14}$$

$$\begin{aligned}
&\textbf{elseif } Q_1[x] \textbf{ then } F'[R_1[x]]\\
&\textbf{elseif } Q_2[x] \textbf{ then } F'[R_2[x]]\\
&\ \dots\\
&\textbf{else } Q_n[x] \textbf{ then } F'[R_n[x]].
\end{aligned}$$

Based on this statement we construct a VCG, which takes as an input program (2) annotated with its specification I_F and O_F, and generates the verification conditions (10), (11), (12) and (13). Moreover, the theorem gives, in fact, two statements, namely:

– *Soundness*: If (10), (11), (12) and (13) hold, then the program (2) is correct, and

– *Completeness*: If (2) is correct, then (10), (11), (12) and (13) hold.

A precise proof of the theorem, based on the fixpoint theory of programs [11], is presented in [7], and completed in [9].

3.3 Proving the Verification Conditions

As we have already said, the coherence check is done at the beginning of the verification process—it is also realized by proving the validity of the respective conditions: (3), (4), (5), (6), (7), (8) and (9). Partial correctness is guarantied by (10), (11), (12), and termination—(13).

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one. The latter one is expressed by using a *simplified version*(14) of the initial program (2), and the condition itself expresses a property of that *simplified version* (13). The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof may be omitted, because different recursive programs may have the same *simplified version.*

Proofs of the verification conditions may be done by using a *Theorema* prover (see, e.g., [3],[5]) or by delivering the proof problem itself to another specialized tool. For serving the termination proofs, actually for omitting the proof redundancy, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof – it only involves matching against simplified versions.

4 Example and Discussion

In order to make clear our experiments, we consider again a powering function P, however we provide this time a different implementation, namely *binary powering*:

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

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:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n. \tag{15}$$

The (automatically generated) conditions for **coherence** are:

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

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow \mathrm{Even}[n]) \tag{17}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow \mathrm{Odd}[n]) \tag{18}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{19}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow \mathbb{T}) \tag{20}$$

One sees that the formulae (16), (19) and (20) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (17) and (18) are easy consequences of the elementary theory of reals and naturals. For the further check of **correctness** the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 1 = x^n) \tag{21}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{22}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2}\ \Rightarrow m = x^n) \tag{23}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{24}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n) \tag{25}$$

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

where

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

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

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

Now, for this buggy version of P we may see that all the respective verification conditions remain the same—and thus the program is correct—except one, namely, (21) is now:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{27}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

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

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (23) is now:

$$(\forall x, n : n \in \mathbb{N})\ (\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{28}$$

which itself reduces to:

$$m = x^{n/2}\ \Rightarrow m = x^n$$

From here, we see that the "Even" branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2}\ \Rightarrow m = x^n$$

which finally leads to the correct version of P.

5 Conclusions

The approach to program verification presented here is a result of an experimental work with the aim of practical verification of recursive programs. Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

References

1. Y. Bertot, P. Casteran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Verlag, 2004.
2. F. Blanqui , S. Hinderer, S. Coupet-Grimal, W Delobel, A. Kroprowski. A Coq library on rewriting and termination. *http://coq.inria.fr/contribs/CoLoR.html*
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. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.
4. B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition, 1981.
5. B. Buchberger, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
6. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.
7. T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2006. To appear.
8. M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.
9. L. Kovacs, N. Popov, T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, Paphos, Cyprus, 2006.
10. J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.
11. Manna, Z.: *Mathematical Theory of Computation.* McGraw-Hill Inc. (1974)
12. PVS: Specification and Verification System. *http://pvs.csl.sri.com*