Verification of Simple Recursive Programs in Theorema: Completeness of the Method

Nikolaj Popov*
RISC – Linz, Austria
popov@risc.uni-linz.ac.at

Abstract

We report work concerning the theoretical basis and the implementation in the Theorema system of a methodology for the generation of verification conditions for recursive procedures, with the aim of practical verification of recursive programs. We develop a method for proving total correctness properties of programs which have simple functional recursive definitions, and we discuss its different aspects. We also define a class of programs for which the method is complete and we give a detailed proof of the completeness statement.

Introduction

We discuss here a practical approach to automatic generation of verification conditions for functional recursive programs and its possible completeness. The implementation is part of the *Theorema* system, and complements the research performed in the *Theorema* group on verification and synthesis of functional and imperative algorithms based on logic principles [1, 2, 7].

We consider the correctness problem expressed as follows: *given* the program (by its source text) for the function F and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input

*The program verification project in the frame of e-Austria Timişoara is supported by BMBWK (Austrian Ministry of Education, Science and Culture), BMWA (Austrian Ministry of Economy and Work) and MEC (Romanian Ministry of Education and Research). The *Theorema* project is supported by FWF (Austrian National Science Foundation) – SFB project P1302.

and the output $O_F[x, y]$, *generate* the verification conditions which are [minimally] sufficient for the program to satisfy the specification.

For simplifying the presentation, we consider here the "homogeneous" case: all functions and predicates are interpreted over the same domain.

Proving the verification conditions will require the *specific theory* relevant to this domain and to the auxiliary functions and predicates which occur in the program. Our work aims at expressing the correctness of the program using only this specific theory, and not a certain theory of computation (which would make automatic reasoning about the verification conditions more difficult). For this we use the fixpoint theory [4, 9, 3, 8] in order to derive the expression of the verification conditions for a whole class of programs.

By correctness of the program we mean both *partial correctness* (1) and *termination* (2):

$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge F[x] \downarrow \Longrightarrow O_F[x, F[x]]), \tag{1}$$

$$(\forall x \in \mathbb{D})\ (I_F[x] \Longrightarrow F[x] \downarrow), \tag{2}$$

where the $\downarrow$ is the unary predicate symbol expressing termination (see below section Completeness of the Method). As one sees, (3) is a logical consequence of (1) and (2), but not the vice versa (for example when I_F and O_F are *true* but F never terminates, i.e., $F[x] \downarrow$ is *false* for any x).

$$(\forall x \in \mathbb{D})\ (I_F[x] \Longrightarrow O_F[x, F[x]]), \tag{3}$$

In order for the program to be correct, the correctness formula (3) must be a logical consequence of the formulae corresponding to the definition of the function (and the specific theory). Additionally, one needs to ensure termination, which we study for a certain class of problems.

The method we are presenting here takes a recursive program having a certain shape and its specification and generates several verification conditions. If the verification conditions hold, then the program is totally correct with respect to its specification. Therefore we say that the method is sound, that is the validity of the verification conditions implies the correctness of the program with respect to the specification.

The method would be complete if the correctness of the program with respect to the specification, implies the validity of the verification conditions (generated by the method). Completeness of the method is important, because it leads to an early detection of bugs (when a proof of a verification condition fails). Starting from a program and its specification, we generate the verification conditions and try to prove them (by using

Theorema). If we manage to prove all of them, we are done. In case we do not manage to prove some of the verification conditions (the formulae are not true), we do not know whether or not the program is correct. However, if the method is complete then even one verification condition which is not true would mean that the program is not correct and so, something must be changed.

Note that we do not discuss here the soundness or completeness of the proof system which would be used for investigating the validity of the generated verification conditions.

Correct Programs

In this section we state the principles we use for writing correct programs. Although, they are not our invention (similar ideas appear in [5]), we list them here because we want to formalize them.

Building up correct programs: We start from basic functions e.g. addition, multiplication, etc. We define new functions in terms of already known (defined previously) functions by giving their source code, the specification (input and output predicates) and prove their total correctness with respect to the specification (prove the verification conditions).

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 definition

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

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 $O_P[x,n,P[x,n]] \iff P[x,n] = x^n$. Then we prove the correctness of P by proving the verification conditions. Later, after using the powering function P for defining other functions, we decide to replace its definition by another one. Having the same specification, the only thing we should do is to prove that the new definition of P meets the old specification. All the calls made to P preserve their properties.

Appropriate values for the auxiliary functions: In order for the program to be correct, we ensure that the calls to the auxiliary functions have appropriate (in the sense of the specifications) values. For example, we define the following function:

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

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:

$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge Q[x] \implies I_H[x])$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge \neg Q[x] \implies I_G[x])$$

The Method

Let $\mathbb{D}$ be an arbitrary domain. We consider the class of simple recursive programs defined by the program scheme:

$$(\forall x \in \mathbb{D})\ F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]], \tag{4}$$

where Q is a predicate on $\mathbb{D}$ and S, C, and R are auxiliary functions (S is a "simple" function, C is a "combinator" function, and R is a "reduction" function).

In this scheme, the functions F, S and R and the predicate Q are of arity 1 and the function C of arity 2. However, if $\mathbb{D}$ is a vector domain, then different arities will be treated by the same method (see the example of *rem* (13)).

Here we present the method for the automatic generation of verification conditions. The detailed proof of the soundness statement [6] is based on the fixpoint theory of functions.

Let $I_F[x], O_F[x, y]$ be the precondition and the postcondition of F, and similarly let $I_S[x], O_S[x, y], I_C[x, y], O_C[x, y, z], I_R[x], O_R[x, y]$ be the preconditions and the postconditions of the auxiliary functions (we assume that the correctness formula holds for each of them). The (automatically) generated verification conditions for the *total correctness* of the function F are:

$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge \neg Q[x] \implies I_F[R[x]]) \tag{5}$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge Q[x] \implies I_S[x]) \tag{6}$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge \neg Q[x] \implies I_R[x]) \tag{7}$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \neg Q[x] \wedge O_F[R[x], F[R[x]]] \implies I_C[x, F[R[x]]]) \tag{8}$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge Q[x] \implies O_F[x, S[x]]) \tag{9}$$
$$(\forall x \in \mathbb{D})\ (I_F[x] \wedge \neg Q[x] \wedge O_F[R[x], F[R[x]]] \implies O_F[x, C[x, F[R[x]]]]) \tag{10}$$

$$(\forall x \in \mathbb{D})\ (I_F[x] \implies F'[x]\downarrow), \tag{11}$$

where

$$(\forall x \in \mathbb{D})\ F'[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } F'[R[x]]. \tag{12}$$

For better understanding, we give here an example of a program and its specification and we show the verification conditions.

Consider the following program:

$$rem[x,y] = \textbf{ If } x < y \textbf{ then } x \textbf{ else } rem[x-y,y] \tag{13}$$

The domain $\mathbb{D} = \mathbb{N}^2$ and the specification of rem is: the precondition

$$I_{rem}[x,y] \iff y > 0$$

and the postcondition

$$O_{rem}[x,y,z] \iff (\exists q \in \mathbb{N})(x = z + y * q \wedge z < y).$$

The preconditions of all the auxiliary functions are $\mathbb{T}$ (the logical constant *true*).

The (automatically) generated verification conditions for the *total correctness* of the function rem are:

$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x < y \implies (\exists q \in \mathbb{N})(x = x + y * q \wedge x < y))$
$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x \geq y \implies (y > 0))$
$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x \geq y \wedge (\exists q \in \mathbb{N})(x - y = f[x,y] + y * q \wedge b < y) \implies$
$\quad (\exists q \in \mathbb{N})(x = f[x,y] + y * q \wedge f[x,y] < y))$
$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x < y \implies \mathbb{T})$
$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x \geq y \implies \mathbb{T})$
$(\forall x, y \in \mathbb{N})\ (y > 0 \wedge x \geq y \wedge (\exists q \in \mathbb{N})$
$\quad (x - y = f[x,y] + y * q \wedge f[x,y] < y) \implies \mathbb{T}))$
$(\forall x, y \in \mathbb{N})(y > 0 \implies F'[x,y]\downarrow),$

where

$$F'[x,y] = \textbf{ If } x < y \textbf{ then } 0 \textbf{ else } F'[x-y,y].$$

If we manage to prove the verification conditions, then we are sure that the program meets its specification.

The method is not complete. In fact, there can be no complete method for the generation of verification conditions [8]. We illustrate the incompleteness by giving an example for a program which is correct with respect to the specification, but one of the verification conditions does not hold.

Consider the following program:

$$fac[n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } fac[n-1] * n \tag{14}$$

The specification of fac is:
The domain $\mathbb{D} = \mathbb{R}$, precondition $I_{fac}[n] \iff n = 3$ and a postcondition $O_{fac}[n, fac[n]] \iff fac[n] = 6$).

The program fac meets its specification, because for the only possible input 3, the output is 6.

However, the verification condition corresponding to (5) does not hold:

$$(\forall n \in \mathbb{N})\ (n = 3 \wedge \neg n = 0 \implies n = 2). \tag{15}$$

Completeness of the Method

In this section we define a class of programs (coherent) and we prove that if a coherent program is correct with respect to the specification then the verification conditions hold.

Definition. A program as in (4) is *coherent* iff (5), (6), (7) and (8) hold.

From the intuitive point of view, these are programs which satisfy the natural property that the arguments of the functions satisfy the input conditions of those functions.

The following theorem gives the completeness of the method.

Theorem. If a coherent program is correct with respect to the specification, then the verification conditions hold.

Proof. Assume that the program F is coherent, i.e., (5), (6), (7) and (8) hold and is correct, i.e., (2) and (3) hold.

We will show that the verification conditions (5) - (11) hold as well. Here we actually need to show only the validity of (9), (10) and (11) because the others are assumed.

Proving (9): Assume that $I_F[x]$ and $Q[x]$. From this, by the definition of F we obtain $F[x] = S[x]$, from which by the specification of F follows $O_F[x, S[x]]$.

Proving (10): Assume that $I_F[x]$ and $\neg Q[x]$ and $O_F[R[x], F[R[x]]]$. From $I_F[x]$ by (4) we obtain $I_F[R[x]]$. From $I_F[x]$ and $\neg Q[x]$ by the specification we obtain $O_F[x, F[x]]$ and by the definition of F follows $O_F[x, C[x, F[R[x]]]$. From $I_F[R[x]]$ we obtain $O_F[R[x], F[R[x]]]$. From here and $O_F[x, C[x, F[R[x]]]$ follows
$O_F[x, C[x, F[R[x]]]]$.

The proof of the formula (11) is done in the next paragraph.

We remind the main elements of the fixpoint theory. The domain $\mathbb{D}$ is extended to a new domain $\mathbb{D} \cup \{\bot\}$, where $\bot$ is an additional symbol for expressing the nonterminating constant. In the extended domain the programs F are functions from $\mathbb{D} \cup \{\bot\} \to \mathbb{D} \cup \{\bot\}$. The predicates Q which are used in the programs, become functions from $\mathbb{D} \cup \{\bot\} \to \{\mathbb{T}, \mathbb{F}\} \cup \{\bot\}$. The unary predicate $\downarrow$ is defined by the formula: $(\forall n)(n \downarrow \Longleftrightarrow n \neq \bot)$. Here we make the natural assumption: For any function F, $F[\bot] = \bot$.

In addition we will need the nowhere defined function Ω which is defined as $(\forall d)(\Omega[d] = \bot)$.

We will use the restricted graph function defined as

$$(\forall f)(RGraph[f] = \{\langle x, y\rangle : f[x] = y \wedge y \downarrow\})$$

and functions are identified with their *RGraph*s.

We consider the operator Γ associated to the program (4):

$$\lambda F.\lambda x.\ \textbf{If}\ Q[x]\ \textbf{then}\ S[x]\ \textbf{else}\ C[x, F[R[x]]]. \tag{16}$$

The function defined by a program is the least fixpoint of the associated operator. From the proof of the Knaster-Tarski Fixpoint theorem [8], we know that the least fixpoint can be constructed as the union of all the finite approximations.

The finite approximations of Γ are defined recursively as follows:

$$f_0 = \Omega$$

$$f_n = \Gamma[f_{n-1}].$$

The least fixpoint F_Γ of Γ is defined as the union of the finite approximations

$$F_\Gamma = \bigcup_n f_n. \tag{17}$$

Lemma 1. $(\forall x \in \mathbb{D})\ (I_F[x] \implies (\exists n \in \mathbb{N})(Q[R^n[x]]))$.

Proof. We proceed by contradiction. Assume that $a \in \mathbb{D}$ and $I_F[a]$. From here by (2) we obtain $F[a] \downarrow$. Assume also $(\forall n \in \mathbb{N})\ (\neg Q[R^n[a]])$.

From this we obtain that $F[a] \neq \bot$ and hence $(\exists b \in \mathbb{D})(\langle a, b\rangle \in RGraph[F])$.

Let f_0, f_1, ... f_m be the finite approximations of F, i.e., $F = \bigcup_i f_i$.

From $\langle a, b\rangle \in RGraph[F] = RGraph[\bigcup_i f_i]$ follows $(\exists k > 0)(\langle a, b\rangle \in RGraph[f_k])$ ($k > 0$ because $a, b \neq \bot$). From here, we obtain that $f_k[a] = b$ which implies $f_k[a] \neq \bot$. By the definition $f_k[a] =$ **If** $Q[a]$ **then** $S[a]$ **else** $C[a, f_{k-1}[R[a]]]$. From $(\neg Q[a])$ (it is so,

because we have $(\forall n \in \mathbb{N})(\neg Q[R^n[a]]))$, we obtain $f_k[a] = C[a, f_{k-1}[R[a]]]$. Since $f_k[a] \downarrow$, we have $f_{k-1}[R[a]] \downarrow$. Applying the same reason k times, we obtain that $f_{k-k}[R^k[a]] \downarrow$. This contradicts to the definition of $f_0 = \Omega$.

Lemma 2. $(\forall x \in \mathbb{D})\ (I_F[x] \implies (\exists n \in \mathbb{N})(Q[R^n[x]] \implies F'[x] \downarrow))$.

Proof. Assume that $x_0 \in \mathbb{D}$, $I_F[x_0]$ and $Q[R^n[x_0]]$ for some $n \in \mathbb{N}$.

1) case. $n = 0$. This means that $Q[x_0]$ and hence $F'[x_0] = 0$, which implies $F'[x] \downarrow$.

2) case. $n > 0$. Assume that n is the first such that $Q[R^n[x_0]]$, that is $\neg Q[R^i[x_0]]$ for any $0 \leq i < n$. From $I_F[x_0]$ and $\neg Q[R[x_0]]$ by (7) and (5) we obtain $I_R[x_0]$ and $I_F[R[x_0]]$. In the same fashion we obtain $R^i[x_0] \downarrow$ and $I_F[R^i[x_0]]$ for any $0 \leq i < n$. By expanding the definition of F' we obtain $F'[x_0] = F'[R[x_0]] = ... = F'[R^n[x_0]] = 0$, which implies $F'[x_0] \downarrow$.

This completes the proofs of lemma 2 and the theorem.

The completeness of the method helps debugging programs in the following way:
Assume that some of the verification conditions (9) – (11) do not hold. Then first check if the program is coherent, i.e., check the validity of (5) – (8).
If yes, i.e., (9) or (10) or (11) does not hold, then the program does not meet its specification.

Conclusion and Further Work

In order to be able to define a *complete* method for the generation of verification conditions, we introduce the notion of *coherent* programs. These are programs which satisfy the natural property that the arguments of the auxiliary functions satisfy the input conditions of those functions.

Part of the methods described here are implemented in the *Theorema* system and we are studying various test cases in order to improve the power of our condition generator, in particular by generalizing this method to more complex program schemes. Moreover, the concrete proof problems are used as test cases for our provers and for experimenting with the organization of the mathematical knowledge.

References

[1] A. Craciun; B. Buchberger. Functional Program Verification with Theorema. In *CAVIS-03 (Computer Aided Verification of Information Systems)*, Institute e-Austria Timisoara, February 2003.

[2] B. Buchberger. Verified Algorithm Development by Lazy Thinking. In *IMS 2003 (International Mathematica Symposium)*, Imperial College, London, July 2003.

[3] C. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[4] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.

[5] M. Kaufmann; J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.

[6] N. Popov. Verification of Simple Recursive Programs: Sufficient Conditions. Technical Report 04-06, RISC-Linz, Austria, 2004.

[7] T. Jebelean; L. Kovacs; N. Popov. Verification of Imperative Programs in Theorema. In *1st South-East European Workshop in Formal Methods (SEEFM03)*, 2003. Thessaloniki, Greece, 20 November 2003.

[8] J. Loeckx; K. Sieber. *The Foundations of Program Verification*. Teubner, second edition, 1987.

[9] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach*. MIT Press, 1977.