Proving Partial Correctness and Termination of Mutually Recursive Programs

Nikolaj Popov and Tudor Jebelean
Research Institute for Symbolic Computation
Johannes Kepler University of Linz
Austria
Email: {popov,jebelean}@risc.uni-linz.ac.at

***Abstract*—We present an environment for proving correctness of mutually recursive functional programs. 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.**

I. Introduction

We develop a method for the generation of verification conditions for proving correctness of mutually recursive functional programs. Our focus is on the generation of the conditions, and we do not treat here the problem of discharging them. We assume that the specification and the program are provided and our task is to generate sound and complete verification conditions. (In fact, we believe that specifications should be developed before the program is written.)

Mutual recursion is a special form of recursion where two programs are defined in terms of each other. Moreover, the two programs form a system, and its solution is the computable function which is defined by the programs.

We study the class of simple mutually recursive programs and we extract the purely logical conditions which are sufficient for the program correctness. They are inferred using Scott induction and induction on natural numbers in the fixpoint theory of functions and constitute a meta-theorem.

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 [2]. 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 [5].

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, ... , 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, and the condition $O_F[x, F[x]]$ holds: 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, ... , VC_n hold in the theory $Th[D]$ of the domain D, then the program F satisfies its specification I_F and O_F.

A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold. In fact, if the VCG is incomplete then some verification conditions may not hold although the program is correct.

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.

This work is performed in the frame of the *Theorema* system [1], 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 and 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. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.).

II. Coherent Programs

In this section we state the general 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

[3]), we state them here because we want to emphasize and later formalize them. Similar ideas appear also in software engineering—they are called there *Design by Contract* or *Programming by Contract* [4].

We build our system such that it preserves the modularity principle, that is, each concrete implementation of a program may be replaced by another one at any time.

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 correctness with respect to the specification.

The next property 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 us the possibility of easy replacement of existing functions.

In order to achieve the modularity, 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 function calls.*

We now define the class of coherent programs as those which obey the *appropriate values to the function calls* principle. The general definition comes in two parts: for functions defined by composition and for functions defined by *if-then-else*.

Definition 2.1: Let F be obtained from H, G_1, ..., G_n by composition:

$$F[x] = H[G_1[x], \ldots, G_n[x]]. \tag{1}$$

The program F with the specification (I_F and O_F) is *coherent* with respect to its auxiliary functions H, G_i and their specifications (I_H and O_H), (I_{G_i} and O_{G_i}) *if and only if*

$$(\forall x : I_F[x]) \implies I_{G_1}[x] \wedge \ldots \wedge I_{G_n}[x] \tag{2}$$

and

$$(\forall x : I_F[x])\ (\forall y_1 \ldots y_n)\ (O_{G_1}[x, y_1] \wedge \ldots \wedge O_{G_n}[x, y_n] \implies \implies I_H[y_1, \ldots, y_n]). \tag{3}$$

Definition 2.2: Let F be obtained from H, G by if-then-else:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x]. \tag{4}$$

The program F with the specification (I_F and O_F) is *coherent* with respect to its auxiliary functions H, G and their specifications (I_H and O_H), (I_G and O_G)

if and only if

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x]) \tag{5}$$

and

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]). \tag{6}$$

Note that H and G may contain invocations of the main program F in their definitions, however, this would be treated as a combination of *if-then-else* and a *composition*. The predicate Q does not contain any invocation of F.

As a first step of the verification process, before going to the real verification, we check if the program is coherent. It is not that programs which are not coherent are necessarily not correct. However, if we want to achieve the modularity of our system, we need to restrict to dealing only with coherent programs.

III. Generation of Verification Conditions

Simple Mutually Recursive Programs are the simplest mutually recursive programs, however their study gives an impression how one can perform verification in more general setting.

We look at programs F, defined by the system F_1, F_2:

$$F[x] = F_1[x], \tag{7}$$

where:

$$F_1[x] = \textbf{ If } Q_1[x] \textbf{ then } S_1[x] \textbf{ else } C_1[x, F_2[R_1[x]]], \tag{8}$$

and

$$F_2[x] = \textbf{ If } Q_2[x] \textbf{ then } S_2[x] \textbf{ else } C_2[x, F_1[R_2[x]]], \tag{9}$$

where Q_1 and Q_2 are predicates and $S_1, S_2, C_1, C_2, R_1, R_2$ are auxiliary functions. Their names are chosen such that, $S_i[x]$ is a "simple" function, $C_i[x, y]$ is a "combinator" function, and $R_i[x]$ is a "reduction" function. We assume that the functions S_i, C_i, and R_i satisfy their specifications given by $I_{S_i}[x]$, $O_{S_i}[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$.

The specifications of the two functions F_1 and F_2 are given, that is, $I_{F_1}[x]$, $O_{F_1}[x, y]$ and $I_{F_2}[x]$, $O_{F_2}[x, y]$. The specification of the main function F is (by definition) the same as the specification of F_1.

A. Coherent Simple Mutually Recursive Programs

In order to perform the coherence check, we define here the relevant verification conditions, which are derived from the definition of coherent programs 2.1 and 2.2, namely:

Definition 3.1: Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Then the program F as defined in 7 with its specification (I_{F_1}, O_{F_1}) is coherent if F_1, F_2 are coherent with respect to S_i, C_i, R_i, and their specifications, if and only if the following conditions hold:

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies I_{S_1}[x]) \tag{10}$$

$$(\forall x : I_{F_1}[x])\ (\neg Q_1[x] \implies I_{F_2}[R_1[x]]) \tag{11}$$

$$(\forall x : I_{F_1}[x])\ (\neg Q_1[x] \implies I_{R_1}[x]) \tag{12}$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x] \wedge O_{F_2}[R_1[x], y] \implies I_{C_1}[x, y]) \quad (13)$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies I_{S_2}[x]) \quad (14)$$

$$(\forall x : I_{F_2}[x])\ (\neg Q_2[x] \implies I_{F_1}[R_2[x]]) \quad (15)$$

$$(\forall x : I_{F_2}[x])\ (\neg Q_2[x] \implies I_{R_2}[x]) \quad (16)$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x] \wedge O_{F_1}[R_2[x], y] \implies I_{C_2}[x, y]) \quad (17)$$

As we can see, the above conditions correspond very much to our intuition about coherent programs, namely:

- 10 treats the special case in F_1, that is, $Q_1[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S_1.
- 11 treats the general case in F_1, that is, $\neg Q_1[x]$ holds and a call to F_2 is applied, thus the new input $R_1[x]$ must fulfill the precondition of F_2.
- etcetera

B. Verification Conditions and their Soundness

If all the generated verification conditions hold as logical formulae in the theory of the domain on which the program is defined, then the program is correct with respect to its specification. The latter statement we call *Soundness* theorem, and we are now ready to define it for the class of coherent simple mutually recursive programs.

Theorem 3.1: Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Let also the simple mutually recursive program F as defined in 7 with its specification (I_{F_1}, O_{F_1}) be coherent, that is, F_1, F_2 be coherent with respect to S_i, C_i, R_i, and their specifications. Then F is totally correct with respect to (I_{F_1}, O_{F_1}) if the following verification conditions hold:

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies O_{F_1}[x, S_1[x]]) \quad (18)$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x] \wedge O_{F_2}[R_1[x], y] \implies \implies O_{F_1}[x, C_1[x, y]]) \quad (19)$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies O_{F_2}[x, S_2[x]]) \quad (20)$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x] \wedge O_{F_1}[R_2[x], y] \implies \implies O_{F_2}[x, C_2[x, y]]) \quad (21)$$

$$(\forall x : I_{F_1}[x])\ (F_1'[x] = \mathbb{T}) \quad (22)$$

where:

$$F_1'[x] = \textbf{ If } Q_1[x] \textbf{ then } \mathbb{T} \textbf{ else } F_2'[R_1[x]] \quad (23)$$

$$F_2'[x] = \textbf{ If } Q_2[x] \textbf{ then } \mathbb{T} \textbf{ else } F_1'[R_2[x]]. \quad (24)$$

As we can see, the above conditions constitute the following principle:

- 18, 20 prove that the base cases for F_1 and F_2 are correct.
- 19, 21 prove that the recursive expressions for F_1 and F_2 are correct under the assumption that the reduced calls are correct.
- 22 prove that a simplified version F_1' of F_1, (whose definition also involves a simplified version F_2' of F_2) terminates for all possible inputs x: $I_{F_1}[x]$.

The proof of the *Soundness* statement is split into two major parts: prove partial correctness using Scott induction; prove termination. We skip the detailed proof for a lack of space – a comprehensive expose would need several pages.

C. Completeness of the Verification Conditions

Theorem 3.2: Let S_i, C_i, and R_i (for $i = 1, 2$) be functions which satisfy their specifications (I_{S_i}, O_{S_i}), (I_{C_i}, O_{C_i}), and (I_{R_i}, O_{R_i}). Let also the simple mutually recursive program F as defined in 7 with its specification (I_{F_1}, O_{F_1}) be coherent, that is, F_1, F_2 be coherent with respect to S_i, C_i, R_i, and their specifications, and the output specifications of F_i, (O_{F_i}) are functional ones.

Then if F_1 and F_2 are totally correct with respect to (I_{F_i}, O_{F_i}) then the following verification conditions hold:

$$(\forall x : I_{F_1}[x])\ (Q_1[x] \implies O_{F_1}[x, S_1[x]]) \quad (25)$$

$$(\forall x, y : I_{F_1}[x])\ (\neg Q_1[x] \wedge O_{F_2}[R_1[x], y] \implies \implies O_{F_1}[x, C_1[x, y]]) \quad (26)$$

$$(\forall x : I_{F_2}[x])\ (Q_2[x] \implies O_{F_2}[x, S_2[x]]) \quad (27)$$

$$(\forall x, y : I_{F_2}[x])\ (\neg Q_2[x] \wedge O_{F_1}[R_2[x], y] \implies \implies O_{F_2}[x, C_2[x, y]]) \quad (28)$$

$$(\forall x : I_{F_1}[x])\ (F_1'[x] = \mathbb{T}) \quad (29)$$

where:

$$F_1'[x] = \textbf{ If } Q_1[x] \textbf{ then } \mathbb{T} \textbf{ else } F_2'[R_1[x]] \quad (30)$$

$$F_2'[x] = \textbf{ If } Q_2[x] \textbf{ then } \mathbb{T} \textbf{ else } F_1'[R_2[x]]. \quad (31)$$

which are the same as 18, 19, 20, 21, 22, and 23, 24 from the Soundness theorem 3.1.
Due to lack of space we do not provide the proof here.

IV. Example: Even and Odd

In order to illustrate the class of simple mutually recursive programs, and, actually what are the necessary and sufficient conditions for the program to be correct, we point to an example for checking whether a given natural number is even or not.

Consider the system E, defined with the help of the functions EV and OD for checking whether a given natural number is even or not:

$$E[x] = EV[x], \quad (32)$$

where:

$$EV[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{T} \textbf{ else } OD[x - 1], \quad (33)$$

$$OD[x] = \textbf{ If } x = 0 \textbf{ then } \mathbb{F} \textbf{ else } EV[x - 1], \quad (34)$$

with the specification:

$$(\forall x)\ (I_{EV}[x] \iff x \in \mathbb{N}),$$

$$(\forall x, y)\ (O_{EV}[x, y] \iff$$
$$\iff (Even[x] \ \wedge\ y = \mathbb{T}) \ \vee\ (Odd[x] \ \wedge\ y = \mathbb{F})),$$
$$(\forall x)\ (I_{OD}[x] \iff x \in \mathbb{N}),$$
$$(\forall x, y)\ (O_{OD}[x, y] \iff$$
$$\iff (Even[x] \ \wedge\ y = \mathbb{F}) \ \vee\ (Odd[x] \ \wedge\ y = \mathbb{T})),$$

The program E is supposed to check whether a given natural number is even or not, that is, for any natural number x, if x is an even number, it should return $\mathbb{T}$, and if x is odd, $\mathbb{F}$.

Before starting with the essential part of the verification, we first check if the program E is coherent. In order to perform the coherence check, we instantiate the relevant conditions:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{35}$$
$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{36}$$
$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{37}$$
$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (Even[x - 1] \ \wedge\ y = \mathbb{F}) \vee$$
$$\vee\ (Odd[x - 1] \ \wedge\ y = \mathbb{T}) \implies \mathbb{T}) \tag{38}$$
$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies \mathbb{T}) \tag{39}$$
$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies x - 1 \in \mathbb{N}) \tag{40}$$
$$(\forall x : x \in \mathbb{N})\ (x \neq 0 \implies \mathbb{T}) \tag{41}$$
$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (Even[x - 1] \ \wedge\ y = \mathbb{T}) \vee$$
$$\vee\ (Odd[x - 1] \ \wedge\ y = \mathbb{F}) \implies \mathbb{T}). \tag{42}$$

As we can see, most of the conditions are trivial to prove, because we have $\mathbb{T}$ at the right hand side of an implication. The origin of these $\mathbb{T}$ are the preconditions of some of the auxiliary functions, e.g., the constant function $\lambda x.\mathbb{T}$, the decrement function $\lambda x.x - 1$, etc. In fact, only 36 and 40 require proofs, however, they are easily tractable in the theory of natural numbers. After we are convinced that E is coherent, we instantiate the relevant verification conditions for proving correctness:

$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies$$
$$\implies (Even[x] \ \wedge\ \mathbb{T} = \mathbb{T}) \ \vee\ (Odd[x] \ \wedge\ \mathbb{T} = \mathbb{F})) \tag{43}$$
$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (Even[x - 1] \ \wedge\ y = \mathbb{F}) \vee$$
$$\vee\ (Odd[x - 1] \ \wedge\ y = \mathbb{T}) \implies$$
$$\implies (Even[x] \ \wedge\ y = \mathbb{T}) \ \vee\ (Odd[x] \ \wedge\ y = \mathbb{F})) \tag{44}$$
$$(\forall x : x \in \mathbb{N})\ (x = 0 \implies (Even[x] \ \wedge\ \mathbb{T} = \mathbb{T}) \vee$$
$$\vee\ (Odd[x] \ \wedge\ \mathbb{T} = \mathbb{F})) \tag{45}$$
$$(\forall x, y : x \in \mathbb{N})\ (x \neq 0 \wedge (Even[x - 1] \ \wedge\ y = \mathbb{T}) \vee$$
$$\vee\ (Odd[x - 1] \ \wedge\ y = \mathbb{F}) \implies$$
$$\implies (Even[x] \ \wedge\ y = \mathbb{F}) \ \vee\ (Odd[x] \ \wedge\ y = \mathbb{T})) \tag{46}$$

We see that all the verification conditions are tractable in the theory of natural numbers. Essentially, one has to prove that: if $x \neq 0$ and $Even[x - 1]$ then $Odd[x]$, and, complementary: if $x \neq 0$ and $Odd[x - 1]$ then $Even[x]$.

Now we need to prove the termination of E, that is:

$$(\forall x : x \in \mathbb{N})\ (EV'[x] = \mathbb{T}) \tag{47}$$

where:

$$EV'[x] = \ \textbf{If}\ x = 0\ \textbf{then}\ \mathbb{T}\ \textbf{else}\ OD'[x - 1] \tag{48}$$
$$OD'[x] = \ \textbf{If}\ x = 0\ \textbf{then}\ \mathbb{T}\ \textbf{else}\ EV'[x - 1]. \tag{49}$$

It is now interesting to see, that EV' and OD' have the same definitions (up to renaming), and we can merge into one, namely:

$$EVOD'[x] = \ \textbf{If}\ x = 0\ \textbf{then}\ \mathbb{T}\ \textbf{else}\ EVOD'[x - 1]. \tag{50}$$

After this transformation, the termination condition is as follows:

$$(\forall x : x \in \mathbb{N})\ (EVOD'[x] = \mathbb{T}). \tag{51}$$

For serving the termination proofs, we have developed a specialized method [6], which consists of: generate a simplified version of the main program – in the example this is 50 and then prove its termination 51. In fact, proving may be replaced by checking if its termination proof is in a specialized library. The main idea behind is that many different programs have the same simplified versions and therefore no new proof is needed. For example, the simplified version 50 is the same as the simplified version of any primitive recursive function and the termination condition 51 remains unchanged.

V. Conclusions

In this paper, we defined necessary and sufficient conditions for simple mutually recursive programs to be totally correct. These are expressed by two theorems. However, the concrete proof obligations (verification conditions) are first order predicate logic formulae, which are provable in the theory of the domain on which the program is executed.

References

[1] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.

[2] T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2008. To appear.

[3] 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.

[4] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[5] N. Popov and T. Jebelean. A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, *Proceedings of FORMED'08*, pages 121–130, March 2008. to appear as ENTCS volume, Elsevier.

[6] N. Popov and T. Jebelean. Proving Termination of Recursive Programs by Matching Against Simplified Program Versions and Construction of Specialized Libraries in Theorema. In D. Hofbauer and A. Serebrenik, editors, *Proceedings of 9-th International Workshop on Termination (WST'07)*, pages 48–52, Paris, France, June 2007.