Verification of Functional Programs Containing Nested Recursion

Nikolaj Popov, Tudor Jebelean*

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

Abstract. We present an environment for proving partial correctness of recursive functional programs which contain nested recursive calls. 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. We demonstrate our method on the McCarthy 91 function, which is considered a "challenge problem" for automated program verification.

1 Introduction

We develop a method for the generation of verification conditions for proving partial correctness of recursive functional programs which contain nested recursive calls. Our focus is on the generation of the conditions, and we do not treat here the problem of proving 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.)

Recursive programs are called nested when an argument to a recursive call contains another invocation of the main recursive program.

For example:

$$f[x] = \textbf{ If } x = 0 \textbf{ then } 0 \textbf{ else } f[f[x-1]].$$

We approach the problem of program verification by studying the most frequent program schemata. We have studied so far Simple Recursive [6] and Fibonacci-like [15] schemata.

When deriving necessary (and also sufficient) conditions for program correctness, we actually prove at the meta-level that for any program of a certain class

* The Theorema project is supported by FWF (Austrian National Science Foundation) – SFB project F1302. The program verification project in the frame of e-Austria Timişoara is supported by BMBWK (Austrian Ministry of Education, Science and Culture). Additional support comes from INTAS project Ref. Nr 05-1000008-8144.

(defined by a certain schema) it suffices to check only the respective verification conditions. This is very important for the automation of the whole process, because the production of the verification conditions is not expensive from the computational point of view.

In this paper we study, in particular, a class of recursive functional programs which contain nested recursive definitions and we extract the purely logical conditions which are sufficient for the program correctness.

The logical conditions are inferred using Scott induction [1], [11] in the fixpoint theory of programs and constitute two meta-theorems which are proven once for the whole class. The concrete verification conditions for each program are then provable without having to use the fixpoint theory.

In order to illustrate the method and the class of recursive functions which may contain nested recursive definitions, we presented here the McCarthy 91 function [13], [12], which is considered a "challenge problem" for automated program verification.

We consider the partial 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, if F terminates on x, (we write $F[x] \downarrow$) then the condition $O_F[x, F[x]]$ holds. This is also called "partial correctness" of the program F:

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

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, ... , 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.

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 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.). More details about *Theorema* are available at `www.theorema.org`.

2 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 [8]), 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* [14].

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.

This simple inductively defined principle would guarantee that no wrong program may enter our system. 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 1. *Let F be obtained from H, G_1, ..., G_n by composition:*

$$F[x] = H[G_1[x], \ldots, G_n[x]]. \tag{2}$$

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

$$(\forall x : I_F[x]) \implies I_{G_1}[x] \wedge \ldots \wedge I_{G_n}[x] \tag{3}$$

and

$$(\forall x : I_F[x])\ (\forall y_1 \ldots y_n)\ (O_{G_1}[x, y_1] \wedge \cdots \wedge O_{G_n}[x, y_n] \implies I_H[y_1, \ldots, y_n]). \tag{4}$$

Definition 2. *Let F be obtained from H, G by if-then-else:*

$$F[x] = \textbf{If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x]. \tag{5}$$

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

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

and

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]). \tag{7}$$

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.

3 Generation of Verification Conditions

In order to prove partial correctness, we extract the purely logical conditions which are sufficient and also necessary for the program to be partially correct.

By the following schema we define the class of programs which may contain nested recursion, namely we look at programs of the form:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]], \quad (8)$$

where Q is a predicate and S, C_i, R are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_1[x, y]$, $\ldots$, $C_k[x, y]$ are "combinator" functions, and $R[x]$ is a "reduction" function).

We assume that the functions $S, C_1, \ldots, C_k$ and R satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_R[x]$, $O_R[x, y]$.

3.1 Coherent Nested Recursive Programs

We start up with instantiating the definitions for coherent programs (1) and (2), namely:

Definition 3. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Then the program F as defined in* (8) *with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications, if and only if the following conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x]) \quad (9)$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R[x]]) \quad (10)$$

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_R[x]) \quad (11)$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \quad (12)\\ (\neg Q[x] \ \wedge\ O_F[R[x], y_1] \ \wedge\ O_F[y_2, y_3] \ \wedge\ \ldots \ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge \\ \wedge\ O_{C_k}[x, y_1, y_2] \ \wedge\ O_{C_{k-1}}[x, y_3, y_4] \ \wedge\ \ldots \ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \implies \\ I_{C_1}[x, y_1] \ \wedge\ I_{C_2}[x, y_3] \ \wedge\ \ldots \ \wedge\ I_{C_k}[x, y_{2k-1}] \wedge \\ \wedge\ I_F[y_2] \ \wedge\ I_F[y_4] \ \wedge\ \ldots \wedge I_F[y_{2k-2}])\end{gathered}$$

Now the conditions for coherence look a bit complicated, however, in the example we will see that this is not a case. Moreover, our experience shows that proving the conditions for coherence of concrete examples is relatively easy, compared, for example, to proving partial correctness conditions.

Looking closer at the conditions, we see that our intuition about coherent programs is met, namely:

– (9) treats the special case, that is, $Q[x]$ holds and no recursion is applied, thus the input x must fulfill the precondition of S.
– (10) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the first new input $R[x]$ must fulfill the precondition of the main function F.
– (11) treats the general case, that is, $\neg Q[x]$ holds and recursion is applied, thus the input x must fulfill the precondition of the reduction function R.
– (12) treats the general case, and expresses in a cascade manner, that all the inputs to the combinator functions $C_1, \ldots, C_k$ must be appropriate and also all the intermediate inputs to the main function F must be appropriate as well.

After having defined the coherence verification conditions, we go towards defining the verification conditions for ensuring partial correctness.

3.2 Partial Correctness Conditions and their Soundness

We introduce the verification conditions for the class of programs with nested recursion, by providing the relevant *Soundness* theorem.

Theorem 1. *Let for all i, the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (8) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications. Then, F is partially correct with respect to (I_F, O_F) if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{13}$$

$$\begin{gathered}(\forall x, y_1, \ldots, y_{2k} : I_F[x]) \\ (\neg Q[x] \ \wedge\ O_F[R[x], y_1] \ \wedge\ O_F[y_2, y_3] \ \wedge\ \ldots \ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge \\ \wedge\ O_{C_k}[x, y_1, y_2] \ \wedge\ O_{C_{k-1}}[x, y_3, y_4] \ \wedge\ \ldots \ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}] \\ \implies \\ O_F[x, y_{2k}])\end{gathered} \tag{14}$$

The above conditions constitute the following principle:

– (13) prove that the base case is correct.
– (14) prove that the recursive expression is correct under the assumption that all the reduced calls are correct.

Proof:

Using Scott induction, we will show that F is partially correct with respect to its specification, namely:

$$(\forall x : I_F[x])\ (F[x]\downarrow \Longrightarrow O_F[x, F[x]]). \tag{15}$$

We now consider the following partial correctness property ϕ of functions:

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

The first step in Scott induction is to show that ϕ holds for the nowhere defined function Ω (that is, there is no x, such that $\Omega[x]\downarrow$). By the definition of ϕ we obtain:

$$\phi[\Omega] \iff (\forall a)\ (\Omega[a]\downarrow\ \wedge I_F[a] \implies O_F[a, \Omega[a]])),$$

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

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

$$(\forall a)\ (f[a]\downarrow\ \wedge I_F[a] \implies O_F[a, f[a]]), \tag{16}$$

and show $\phi[f_{new}]$, where f_{new} is obtained from f by the main program (8) as follows:

$$f_{new} = \textbf{If } Q[x] \textbf{ then } S[x] \textbf{ else } C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]].$$

Now, we need to show that for an arbitrary a,

$$f_{new}[a]\downarrow\ \wedge I_F[a] \implies O_F[a, f_{new}[a]].$$

Assume $f_{new}[a]\downarrow$ and $I_F[a]$. We have now the following two cases:

- Case 1: $Q[a]$.
  By the definition of f_{new} we obtain $f_{new}[a] = S[a]$ and since $f_{new}[a]\downarrow$, we obtain that $S[a]$ must terminate as well, that is $S[a]\downarrow$. Now using verification condition (13) we may conclude $O_F[a, S[a]]$ and hence $O_F[a, f_{new}[a]]$.
- Case 2: $\neg Q[a]$.
  By the definition of f_{new} we obtain:

  $$f_{new}[a] = C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]$$

  and since $f_{new}[a]\downarrow$, we obtain that all the programs involved in this computation also terminate, that is:

  $$C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]]\downarrow$$

  and say: $C_1[a, f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]] = y_{2k}$,

  $$f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]]\downarrow$$

  and say: $f[C_2[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-1}$,

$$C_2[a, f[\ldots C_k[a, f[R[a]]]]] \downarrow$$

and say: $C_2[a, f[\ldots C_k[a, f[R[a]]]]] = y_{2k-2}$,

$$f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]] \downarrow$$

and say: $f[C_3[a, f[\ldots C_k[a, f[R[a]]]]]] = y_{2k-3}$,

$$\ldots$$

$$f[C_k[a, f[R[a]]]] \downarrow$$

and say: $f[C_k[a, f[R[a]]]] = y_3$,

$$C_k[a, f[R[a]]] \downarrow$$

and say: $C_k[a, f[R[a]]] = y_2$,

$$f[R[a]] \downarrow$$

and say: $f[R[a]] = y_1$, and

$$R[a] \downarrow .$$

From here, by the induction hypothesis, we obtain that:

$$O_F[R[a], y_1] \ \wedge \ O_F[y_2, y_3] \ \wedge \ \ldots \ \wedge \ O_F[y_{2k-2}, y_{2k-1}].$$

On the other hand, by knowing that all the programs C_1, C_2, ..., C_k are partially correct with respect to their specifications, we obtain that:

$$O_{C_1}[a, y_{2k-1}, y_{2k}] \wedge O_{C_2}[a, y_{2k-3}, y_{2k-2}] \wedge \ldots \wedge O_{C_{k-1}}[a, y_3, y_4] \wedge O_{C_k}[a, y_1, y_2].$$

Concerning the verification condition (14), note that all the assumptions from the left part of the implication are at hand and thus we can conclude:

$$O_F[a, y_{2k}],$$

and thus $O_F[a, f_{new}[a]]$.

We conclude that the property ϕ holds for the least fixpoint of (8) and hence, ϕ holds for the function computed by (8), which completes the proof of the soundness theorem (1).

Now we proceed towards the complement of the soundness theorem, namely, the *Completeness* theorem.

3.3 Completeness of the Verification Conditions

Now, we formulate the *Completeness* theorem for the class of programs with nested recursion.

Theorem 2. *Let for all i the functions S, C_i, and R satisfy their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R). Let also the program F as defined in* (8) *with its specification (I_F, O_F) be coherent with respect to S, C_i, R, and their specifications, and the output specifications of F, C_i: (O_F), (O_{C_I}) are functional ones.*

Then, if F is partially correct with respect to (I_F, O_F) then the verification conditions (13) *and* (14) *hold.*

Proof:
We assume now that:

- The functions S, C_i, and R are partially correct with respect to their specifications (I_S, O_S), (I_{C_i}, O_{C_i}), and (I_R, O_R).
- The program F as defined in (8) with its specification (I_F, O_F) is coherent with respect to S, C_i, R, and their specifications.
- The output specifications of F, C_i: O_F, O_{C_i} are functional ones, that is:

$$(\forall x : I_F[x])\ (\exists! y)\ (O_F[x, y]),$$

$$(\forall x, y : I_{C_i}[x, y])\ (\exists! z)\ (O_{C_i}[x, y, z]).$$

- The program F as defined in (8) is partially correct with respect to its specification, that is, the partial correctness formula holds:

$$(\forall x : I_F[x])\ (F[x] \downarrow \implies O_F[x, F[x]]). \tag{17}$$

We show that (13) and (14) are valid as logical formulae by proving them simultaneously.

Take arbitrary but fixed x and assume $I_F[x]$ and $F[x] \downarrow$. We consider the following two cases:

- Case 1: $Q[x]$
  By the definition of F, we have $F[x] = S[x]$, and by using the partial correctness formula (17) of F, we conclude (13) holds. Proving (14) is trivial, because we have $Q[x]$.
- Case 2: $\neg Q[x]$
  Now, proving (13) is trivial. Assume $y_1, \ldots, y_{2k}$ are such that:

$$O_F[R[x], y_1] \ \wedge\ O_F[y_2, y_3] \ \wedge\ \ldots \ \wedge\ O_F[y_{2k-2}, y_{2k-1}] \wedge$$
$$\wedge\ O_{C_k}[x, y_1, y_2] \ \wedge\ O_{C_{k-1}}[x, y_3, y_4] \ \wedge\ \ldots \ \wedge\ O_{C_1}[x, y_{2k-1}, y_{2k}].$$

  Since F is partially correct and $F[x] \downarrow$, we obtain that:

$$C_1[x, F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]]] \downarrow$$

$$F[C_2[x, F[\ldots C_k[x, F[R[x]]]]]] \downarrow$$

$$C_2[x, F[\ldots C_k[x, F[R[x]]]]] \downarrow$$

$$F[C_3[x, F[\ldots C_k[x, F[R[x]]]]]] \downarrow$$

$$\ldots$$

$$F[C_k[x, F[R[x]]]] \downarrow$$

$$C_k[x, F[R[x]]] \downarrow$$

$$F[R[x]] \downarrow$$

$$R[x] \downarrow .$$

From the fact that F is correct follows that it obeys its specification. We assumed $O_f[R[x], y_1]$ holds, and since the output specification is functional, we conclude that $F[R[x]] = y_1$.
Furthermore, since the output specifications of C_i: O_{C_i} are functional predicates, we obtain that:

$$C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k},$$

$$f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-1},$$

$$C_2[x, f[\ldots C_k[x, f[R[x]]]]] = y_{2k-2},$$

$$f[C_3[x, f[\ldots C_k[x, f[R[x]]]]]] = y_{2k-3},$$

$$\ldots$$

$$f[C_k[x, f[R[x]]]] = y_3,$$

$$C_k[x, f[R[x]]] = y_2,$$

$$f[R[x]] = y_1.$$

On the other hand, by the definition of F, we have:

$$F[x] = C_1[x, f[C_2[x, f[\ldots C_k[x, f[R[x]]]]]]]$$

and hence $F[x] = y_{2k}$. Again, from the correctness of F, we obtain:

$$O_F[x, y_{2k}],$$

which had to be proven.

By this we completed our proof of the *Completeness* theorem.

4 Example and Discussion

In order to illustrate the *Soundness* and the *Completeness* theorems, and the class of recursive functions which may contain nested recursive definitions, we consider the McCarthy 91 function, which is considered a "challenge problem" for automated program verification.

The program itself is defined as follows:

$$M[x] = \textbf{ If } x \geq 101 \textbf{ then } x - 10 \textbf{ else } M[M[x + 11]], \tag{18}$$

with the specification:

$$(\forall x)\ (I_M[x] \iff x \in \mathbb{N}) \tag{19}$$

and

$$(\forall x, y)\ (O_M[x, y] \iff (x < 101 \ \wedge\ y = 91) \vee (x \geq 101 \ \wedge\ y = x - 10)). \tag{20}$$

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

$$(\forall x : x \in \mathbb{N})\ (x \geq 101 \ \Rightarrow\ \mathbb{T}) \tag{21}$$

$$(\forall x : x \in \mathbb{N})\ (x \not\geq 101 \ \Rightarrow\ x + 11 \in \mathbb{N}) \tag{22}$$

$$(\forall x : x \in \mathbb{N})\ (x \not\geq 101 \ \Rightarrow\ \mathbb{T}) \tag{23}$$

$$(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \tag{24}$$

$$(x \not\geq 101 \ \wedge\ ((x + 11 < 101 \ \wedge\ y_1 = 91) \vee (x + 11 \geq 101 \ \wedge\ y_1 = x + 11 - 10)) \wedge$$

$$\wedge\ ((y_2 < 101 \ \wedge\ y_3 = 91) \vee (y_2 \geq 101 \ \wedge\ y_3 = y_2 - 10)) \ \wedge\ y_1 = y_2 \ \wedge\ y_3 = y_4$$

$$\Longrightarrow$$

$$\mathbb{T} \ \wedge\ \mathbb{T} \ \wedge\ y_2 \in \mathbb{N} \ \wedge\ y_4 \in \mathbb{N})$$

One sees that the formulae (21) and (23) 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 $x - 10$ ($S[x] = x - 10$) and the projection functions ($C_1[x, y] = y$ and $C_2[x, y] = y$).

The formulae (22) and (24) are easy consequences of the elementary theory of naturals.

For the further check of **correctness** the generated conditions are:

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

$$(x \geq 101 \implies (x < 101 \ \wedge \ x - 10 = 91) \vee (x \geq 101 \ \wedge \ x - 10 = x - 10)).$$

$$(\forall x, y_1, y_2, y_3, y_4 : x \in \mathbb{N}) \tag{26}$$

$$(n \not\geq 101 \ \wedge \ ((x + 11 < 101 \ \wedge \ y_1 = 91) \vee (x + 11 \geq 101 \ \wedge \ y_1 = x + 11 - 10)) \wedge$$

$$\wedge \ ((y_2 < 101 \ \wedge \ y_3 = 91) \vee (y_2 \geq 101 \ \wedge \ y_3 = y_2 - 10)) \ \wedge \ y_1 = y_2 \ \wedge \ y_3 = y_4$$

$$\implies$$

$$(x < 101 \ \wedge \ y_4 = 91) \vee (x \geq 101 \ \wedge \ y_4 = x - 10)).$$

The proofs of these verification conditions are straightforward, and thus the program (18) is partially correct with respect to the specification (19), (20).

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program M is now almost the same as the previous one, but in the base case (when $x \geq 101$) the return value is $x - 11$. The new (wrong) definition of M is:

$$M[x] = \ \textbf{If } x \geq 101 \textbf{ then } x - 11 \textbf{ else } M[M[x + 11]], \tag{27}$$

After generating the verification conditions, we see that all but one are valid, namely:

$$(\forall x : x \in \mathbb{N}) \tag{28}$$

$$(x \geq 101 \implies (x < 101 \ \wedge \ x - 11 = 91) \vee (x \geq 101 \ \wedge \ x - 11 = x - 10)).$$

which reduces to proving:

$$x - 11 = x - 10.$$

Therefore, according to the completeness of the method, we conclude that the program M does not satisfy its specification. Moreover, the failed proof gives a hint for "debugging": we need to change the return value in the case $x \geq 101$ to $x - 10$.

A similar experiment shows, that in fact, the input condition (20), that is $x \in \mathbb{N}$ is too strong, and could be successfully replaced by $x \in \mathbb{Z}$.

5 Related Research

Proving correctness of recursive programs in an automatic manner is considered as being challenge, even without nested recursive definitions. Studying the behavior of such programs begins with classical papers (e.g., [18]) and it is still under consideration.

Proofs exposed in classical books (e.g., [10], [11]) are very comprehensive, however, their orientation is theoretical rather than practical and mechanized. On the other hand, most of the tools for proving program correctness automatically or semiautomatically, do not treat that special case of recursion if they do recursion at all.

In the PVS system [16] the approach is type theoretical and relies on exploration of certain subtyping properties.

The approach presented in [2] puts additional restrictions, namely, recursive programs are examined first to satisfy non-nested recursive definitions and then they may be considered as nested recursion.

In the RRL system [7], a specialized "cover set induction method" is introduced and the nested recursion is treated by it.

In the Lambda system [5], the recursive programs are treated as fixpoint operators, however, it does not extract automatically the inductive obligations that would correspond to the general case in our settings.

The paper [17], presents two methods for dealing with nested recursion, including termination. However, termination conditions must be provided manually.

The main (and also very essential) difference of our approach is that we are able to formulate conditions which are not only sufficient but also necessary in order for the program to be correct.

6 Conclusions

In this paper, we defined necessary and sufficient conditions for programs which may contain nested recursion to be partially correct. These are expressed by two theorems, whose proofs are carried over in the fixpoint theory of programs. 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.

Furthermore, the concrete proof problems are used as test cases for the provers of *Theorema* and for experimenting with the organization and management of mathematical knowledge.

References

1. J. W. de Bakker and D. Scott. A Theory of Programs. In *IBM Seminar*, Vienna, Austria, 1969.

2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.
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, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
5. S. Finn, M. P. Fourman, and J. Longley. Partial Functions in a Total Setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.
6. T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2008. To appear.
7. D. Kapur and M. Subramaniam. Automating Induction over Mutually Recursive Functions. In *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, pages 117–131, London, UK, 1996. Springer-Verlag.
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. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.
12. Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. *Machine Intelligence*, 5:27–37.
13. Z. Manna and A. Pnueli. Formalization of Properties of Functional Programs. *J. ACM*, 17(3):555–569, 1970.
14. Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
15. 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.
16. PVS: Specification and Verification System. *http://pvs.csl.sri.com*
17. Konrad Slind. Another Look at Nested Recursion. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 498–518, London, UK, 2000. Springer-Verlag.
18. W. W. Tait. Nested Recursion. *Mathematische Annalen*, 143(3):236–250, 1961.