Proving Termination of Recursive Programs by Matching Against Simplified Program Versions and Construction of Specialized Libraries in Theorema

Nikolaj Popov, Tudor Jebelean[⋆]

Research Institute for Symbolic Computation, Linz, A–4232 Hagenberg, Austria
popov@risc.uni-linz.ac.at

Abstract. We report work in progress concerning the program verification environment in the *Theorema* system. As part of the system a specialized strategy for proving termination of recursive functional programs is presented. The detailed termination proofs may in many cases be skipped, because the termination conditions are reusable and thus collected in specialized libraries. Enlargement of the libraries is possible by proving termination of each candidate, but also by taking new elements directly from existing libraries.

Introduction.

Proving correctness of recursive programs is still challenging, especially when by correctness is meant total correctness. There are various approaches, however, there is no (and cannot be) general recipe. Termination proofs exposed in classical books (e.g., [8]) are very comprehensive, however, their orientation is theoretical rather than practical. On the other hand there are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [9],[1]), and this is where our contribution falls into.

Termination proofs of individual programs are, in general, *expensive* from the automatic theorem proving point of view. In some cases, program termination, however, may be ensured – and this is our main contribution – by matching against *simplified versions* (of programs) collected in specialized libraries. An idea on how these libraries are constructed is given at the end of the next section.

In our approach, proving total correctness of a program is split into three distinct steps: first – proving coherence, second – proving partial correctness, and third – proving termination. (The combination of the three steps guaranties the total correctness.)

In more detail, we are given a program (by its source text) which computes the function F and we are given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$.

We say that a program is *coherent* if all the calls made to its auxiliary programs are such that the preconditions of the auxiliary programs are not violated. The following example gives an intuition on what we are doing. Let the program for computing F be:

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

where Q is a total predicate and H and G are auxiliary programs. The specification of F is (I_F, O_F) and the specifications of the auxiliary functions are (I_H, O_H) and (I_G, O_G), respectively. The two verification conditions, ensuring that the calls to the auxiliary functions have

[⋆] 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 F1302.

appropriate values, and that F is coherent are:

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

The coherence check – actually, proving the respective verification conditions – is done at the beginning of the verification process. If the program is not coherent it may still be correct, however, its verification would involve knowledge about the concrete implementation of the auxiliary functions. Thus, the modularity (the easy exchange of program implementations) is lost and it is considered to be out of the scope of our system.

Furthermore, partial correctness and termination are expressed as verification conditions which themselves may be proven without taking into account their order. Moreover, as we have shown in [7], a coherent program (of a certain recursive type) is totaly correct if and only if its verification conditions are valid as logical formulae.

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* of the initial program, and the condition itself expresses a property of that *simplified version*. 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 is not always necessary, and this is what we discuss in the next section.

Our work is performed in the frame of the *Theorema* system (an overview could be found in [3]), which is a mathematical computer assistant aiming at supporting all the phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides 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. *Theorema* includes a collection of general as well as specific provers for various interesting domains (e. g., integers, sets, reals, tuples, etc.).

Libraries of Terminating Programs.

In this section we describe the idea of proving termination of recursive programs by creating and exploring libraries of terminating programs, and thus avoiding redundancy of induction proofs. The core idea is that different recursive programs may have the same *simplified version*.

Let us consider the following very simple recursive program for computing the factorial function:

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

with the specification of $Fact$ (*Input:* $I_{Fact}[n] \iff n \in \mathbb{N}$ and *Output:* $O_{Fact}[n, m] \iff n! = m$). The verification condition for the termination of $Fact$ is expressed using a *simplified version* of the initial function:

$$Fact'[n] = \textbf{ If } n = 0 \textbf{ then } 0 \textbf{ else } Fact'[n-1], \tag{2}$$

namely, the verification condition is

$$(\forall n : n \in \mathbb{N})\ (Fact'[n] = 0). \tag{3}$$

More generally, when having a recursive program which may fit to the scheme:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R_1[x]], \ldots, F[R_k[x]]], \tag{4}$$

where Q is a predicate and S, C, R_1, ..., R_k are auxiliary functions whose total correctness is assumed, the corresponding *simplified version* of F is:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } F'[R_1[x]] + \cdots + F'[R_k[x]],$$

which only depends on Q, R_1, ..., R_k. It is obtained by replacing the function S by 0, and the function C by addition (combining the recursive calls). Namely, the termination condition is

$$(\forall x \ : I_F[x]) \ (F'[x] = 0),$$

which must be proven, based on the logical formulae corresponding to the definition of F' and the theory of the domain of Q, R_1, ..., R_k. Moreover, proving that $(\forall x \ : I_F[x]) \ (F'[x] = 0)$ is equivalent to proving termination of $F'[x]$, for all x satisfying $I_F[x]$ (it is so, because if $F'[x]$ terminates it returns 0, and vice versa), which may be used alternatively.

The program scheme (4) is in fact extended to a more general one – it may have more "else" branches with different C functions on each branch, and additionally, on any branch it may have many recursive calls with different R functions.

A soundness (in fact, soundness and completeness) theorem (a program is totaly correct if and only if its verification conditions are valid) concerning recursive schemes with many "else" branches but at most one recursive call on each branch has been proven [7]. As a consequence of that theorem, we know what the necessary (and sufficient) conditions for program correctness are – these are the verification conditions. A new soundness statement concerning schemes which may have many recursive calls on each branch is to be published.

Note, that different recursive programs may have the same *simplified version*. Notably, (2) becomes the *simplified version* of all the unary primitive recursive functions and thus (3) becomes the termination condition of that class. This is easily provable from the definition of primitive recursion.

For serving the termination proofs, 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.

Starting from a small library – actually it is not only one, but more, because each recursive scheme has several domain based libraries – we intend to enlarge it. One way of doing so is by carrying over the whole proof of any new candidate, appearing during a verification process.

Enlargement within a library is also possible by applying special knowledge retrieval. As we have seen, termination depends on the *simplified version* F' and on the input condition I_F. Considering again the factorial example (1), in order to prove its termination we need to prove (3). Assume, now the pair (2),(3) is in our library. We may now strengthen the input condition I_{Fact} and actually produce a new one:

$$I_{F-new}[n] \Longleftrightarrow (n \in \mathbb{N} \wedge n \geq 100).$$

The *simplified version* $Fact'$ remains the same (2) – we did not change the initial program (1), however, the termination condition becomes:

$$(\forall n : n \in \mathbb{N} \wedge n \geq 100) \ (Fact'[n] = 0), \tag{5}$$

and (after proving the validity) we add it to the library. It is easy to see that any new version of a simplified program which is obtained by strengthening the input condition can also be included in the library without further proof. Assume

$$(\forall x \ : I_F[x]) \ (F'[x] = 0)$$

is a member of a library. Then for any "stronger" input condition $I_{F-strng}$, we have:

$$I_{F-strng}[x] \Longrightarrow I_F[x],$$

and thus

$$(\forall x \ : I_{F-strng}[x]) \ (F'[x] = 0).$$

This is of course not the case for weakening the input condition. Consider the following weakening of I_{Fact}:

$$I_{F-real}[n] \Longleftrightarrow (n \in \mathtt{R}),$$

which leads to nontermination of our $Fact'$ as defined in (3), that is:

$$(\forall n \ : n \in \mathtt{R}) \ (Fact'[n] = 0),$$

which is not anymore a valid formula.

Strengthening of input conditions leads to preserving the termination properties and thus enlarging a library without additional proof is possible. However, for a fixed *simplified version*, keeping (and collecting in some cases) the weakest input condition is the most efficient strategy, because then proving the implication from stronger to weaker condition is relatively easier.

Implementation and experiments. The method described above is implemented in the *Theorema* system. The recursive schemes, we have studied so far, are not covering all the possibilities, however, we are extending them to more complex ones, e.g., mutual recursion and nested recursion. Termination proofs may be done by using a *Theorema* prover (see, e.g., [3],[4]). However, delivering the proof problem itself to another specialized tool (e.g., [6],[5]) is also possible. Enlarging the libraries by taking (and adopting) *simplified versions* directly from other libraries (e.g., the *Coq Library* [2]) can be considered as well.

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, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
5. B. Cook, A. Podelski, A. Rybalchenko. Terminator: Beyond safety. In *In Computer-Aided Verification (CAV06), LNCS 4144*, pp. 415–418, Seattle, USA, 2006.
6. N. Hirokawa, A. Middeldorp. Tyrolean Termination Tool. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA), LNCS 3467*, Nara, Japan, 2005.
7. 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.
8. J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.
9. PVS: Specification and Verification System. *http://pvs.csl.sri.com*