Verification of Multiple Choice Functional Programs in Theorema

Nikolaj Popov, Tudor Jebelean*

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

Extended Abstract

We present a novel method for proving total correctness of recursive functional programs defined by a simple scheme (however, the most frequent one in practice). The method provides not only a proof for correct programs, but also a hint on "what is wrong" if the program is not correct.

As usual, program correctness is transformed into a set of first-order predicate logic formulae by a Verification Condition Generator (VCG). As a distinctive feature of our method, these formulae do not refer to a theoretical model for program semantics or program execution, but only to the theory of the domain used in the program.

In more detail, consider a program which computes a function F (over a certain domain), together with a specification given by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$. One says that the program is totally correct with respect to the specification iff the program terminates on any input x satisfying I_F, and, for each such input, the condition $O_F[x, F[x]]$ holds:

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

We consider the class of *Simple Recursive Programs with Multiple Recursive Calls*, that is the ones which may be expressed as:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R_1[x], \ldots, R_n[x]]], \tag{2}$$

where Q is a predicate and $S, C, R_1, \ldots, R_n$ are auxiliary functions ($S[x]$ is a "simple" function, $C[x, y]$ is a "combinator" function, and $R_1[x], \ldots, R_n[x]$ are"reduction" functions). We assume that the functions S, C, and R_1, $\ldots R_n$ satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_C[x, y]$, $O_C[x, y, z]$, $I_{R_1}[x]$, $O_{R_1}[x, y]$, $\ldots$, $I_{R_n}[x]$, $O_{R_n}[x, y]$. Note that functions F with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

In fact, the method presented here works analogously on the more general class of programs containing **Case** statements (**If–then–else** with several cases).

* The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project. The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project F1302.

We are able to prove the *Soundness* of VCG (the verification condition generator), namely: if all formulae produced by VCG on input F, I_F, O_F hold, then the program F satisfies its specification I_F and O_F. (For more details, see [1]).

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".

Coherent Programs.

We state here 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 [2]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs:

We propose the following methodology for building up a collection of correct programs:

– 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, and its specification (input and output predicates) and prove its total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our collection. 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.

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 those programs – we call this: *Appropriate values for the auxiliary functions.*

Now we define *Coherent programs* as all those, which obey the above restrictions.

In addition to the soundness statement, we have proven a completeness statement for the class *Coherent Simple Recursive Programs*. (More details are available at [3]).

References

1. T. Jebelean, L. Kovacs, and N. Popov. Experimental Program Verification in the Theorema System. *STTT*, pages 1–10, 2006. in press.
2. M. Kaufmann; J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.
3. N. Popov. Verification of Simple Recursive Programs in Theorema: Completeness of the Method. Technical Report 05-06, RISC Report Series, University of Linz, Austria, June 2005.