A Complete Method for Algorithm Validation

Nikolaj Popov and Tudor Jebelean [⋆]

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{popov,jebelean}@risc.jku.at

Abstract. We present some novel ideas for proving total correctness of recursive functional programs and we discuss how they may be used for algorithm validation. As usual, correctness (validation) 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 Nevilles algorithm for polynomial interpolation and show how it may be validated 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.

1 Introduction

The main focus of our research is on proving total correctness of recursive functional programs. In order to reason about programs, one first translates the problem of proving program property into a problem of proving logical assertions, and then prove the assertions. A common way for making such translation is by some verification condition generator—its name is self-explanatory. 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.

Let us say, the program is F and the specification I_F (input predicate), and O_F (output predicate) is provided. The verification conditions generated by VCG are: $VC_1, VC_2, \ldots, VC_n$. After having the verification

[⋆] This research is supported by BMBWK (Austrian Ministry of Education, Science, and Culture) and Upper Austrian Government Project "Technologietransferaktivitäten".

conditions at hand, one has to prove them as logical formulae in the theory of the domain on which the program F is defined, e.g., integers, reals, etc.

Normally, these conditions are given to an automatic or semi-automatic theorem prover. If all of them hold, then the program is correct with respect to its specification. The latter statement we call *Soundness* of the VCG, namely:

Given a program F and a specification I_F (input condition), and O_F (output condition), if the verification conditions generated by the VCG hold as logical formulae, then the program F is correct with respect to the specification $\langle I_F, O_F \rangle$.

It is clear that whenever one defines a VCG, the first task to be done is proving its soundness statement—otherwise it would not be properly called: Verification Condition Generator.

Completing the notion of *Soundness* of a VCG, we introduce its dual—*Completeness* [1]. The respective *Completeness* statement of the VCG is:

Given a program F and a specification I_F (input condition), and O_F (output condition), if the program F is correct with respect to the specification $\langle I_F, O_F \rangle$, then the verification conditions generated by the VCG hold as logical formulae.

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.

2 Automatic Validation of Neville's Algorithm

Using techniques from program verification in systematic mathematical theory exploration is, in our opinion, very promising approach. In particular, when inventing algorithms, one can obtain the exact set of necessary and sufficient conditions for their correctness. On one hand, the help comes with the automatically obtained correctness proof. On the other hand, the inventor may try to prove the correctness of any conjecture, and in case of a failure obtain a counterexample, which may eventually help making a new conjecture.

In order to make the point clear, we demonstrate our method on Neville's algorithm for polynomial interpolation [2], [3] and show how it may be validated fully automatically.

Given n points, there is a unique polynomial of degree $n-1$ which goes through the given points. Neville's algorithm constructs this polynomial.

The original problem is as follows: Given a field K, two non-empty tuples $\overline{x}$ and $\overline{a}$ over K of same length n, such that

$$(\forall i, j \ : \ i, j = 1, \ldots, n) \ (i \neq j \Rightarrow x_i \neq x_j),$$

that is, no two x_i from $\overline{x}$ are the same.

Find a polynomial p over the field K, such that

– $deg[p] \leq n-1$ and
– $(\forall i \ : \ i = 1, \ldots, n) \ (Eval[p, x_i] = a_i)$,

where the $Eval$ function evaluates a polynomial p at value x_i.

This original problem, as stated here, was solved by E. H. Neville [4] by inventing an algorithm for the construction of such a polynomial [5]. The algorithm itself may be formulated as follows:

$$p[\overline{x}, \overline{a}] = \quad \textbf{If } \|\overline{a}\| \leq 1 \quad \textbf{then } Fst[\overline{a}] \tag{1}$$

$$\textbf{else } \frac{(\mathcal{X} - Fst[\overline{x}])(p[Tl[\overline{x}], Tl[\overline{a}]]) - (\mathcal{X} - Lst[\overline{x}])(p[Bgn[\overline{x}], Bgn[\overline{a}]])}{Lst[\overline{x}] - Fst[\overline{x}]},$$

where we use the following notation: $\|\overline{a}\|$ gives the number n of elements of $\overline{a}$; $Fst[\overline{a}]$ gives the first element a_1 of $\overline{a}$; $Lst[\overline{a}]$ gives the last element a_n of $\overline{a}$, provided $\|\overline{a}\| = n$; $Tl[\overline{a}]$ gives the tail of $\overline{a}$, that is, $\overline{a}$ without its first element; $Bgn[\overline{a}]$ gives the beginning of $\overline{a}$, that is, $\overline{a}$ without its last element, and $\mathcal{X}$ is a constant expressing the single polynomial of degree 1, leading coefficient 1 and free coefficient 0.

In order to illustrate how Neville's algorithm works, we consider the following example: $\overline{x} = \langle -1, 0, 1 \rangle$ and $\overline{a} = \langle 3, 4, 7 \rangle$. After executing (1), we obtain:

$$p[\langle -1, 0, 1 \rangle, \ \langle 3, 4, 7 \rangle] = \cdots = \mathcal{X}^2 + 2\mathcal{X} + 4.$$

This polynomial has a degree 2, as expected, and if we now evaluate it at the values -1, 0, and 1, we obtain:

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, -1] = 3,$$

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, 0] = 4,$$

and

$$Eval[\mathcal{X}^2 + 2\mathcal{X} + 4, 1] = 7,$$

which corresponds to the initial $\overline{a}$.

However, in order to be sure that this algorithm would always return the correct polynomial, one has to prove its correctness, and this was done by Neville himself [5].

Our contribution consists in automating the process of the correctness proof. Moreover, as said before, even if a small part of the specification is missing, which sometimes happens, the algorithm would not be correct anymore, and relevant counterexamples may be generated.

Generating automatically the verification conditions, we obtain essentially (omitting only some details) the following formulae:

$$\cdots \wedge IsPoly[p_1] \wedge IsPoly[p_2] \Rightarrow IsPoly[\frac{(\mathcal{X} - Fst[x])p_1 - (\mathcal{X} - Lst[x])p_2}{Lst[x] - Fst[x]}] \tag{2}$$

$$\begin{gathered} \cdots \wedge (\forall i)(i = 1 \ldots \|Tl[x]\|)(Eval[p_1, Tl[x]_i]) = Tl[a]_i \wedge \\ \wedge\ (\forall i)(i = 1 \ldots \|Bgn[x]\|)(Eval[p_2, Bgn[x]_i]) = Tl[a]_i \Rightarrow \\ \Rightarrow (\forall i)(i = 1 \ldots \|a\|)(Eval[\frac{(\mathcal{X} - Fst[x])p_1 - (\mathcal{X} - Lst[x])p_2}{Lst[x] - Fst[x]}, x_i] = a_i) \end{gathered} \tag{3}$$

$$\begin{gathered} \cdots \wedge\ deg[p_1] \leq \|Tl[a]\| - 1 \wedge\ deg[p_2] \leq \|Bgn[a]\| - 1 \Rightarrow \\ \Rightarrow\ deg[\frac{(\mathcal{X} - Fst[x])p_1 - (\mathcal{X} - Lst[x])p_2}{Lst[x] - Fst[x]} \leq \|a\| - 1 \end{gathered} \tag{4}$$

Although the above formulae appear to be very complicated, they are tractable in the theory of polynomials. For example, in (2) we need to prove that for any p_1 and p_2 which are polynomials,

$$IsPoly[\frac{(\mathcal{X} - Fst[x])p_1 - (\mathcal{X} - Lst[x])p_2}{Lst[x] - Fst[x]}]$$

is a polynomial as well. Note that $Fst[x]$ and $Lst[x]$ are constants and $\mathcal{X}$ is a polynomial.

3 Conclusions

In order for a human mathematician to increase the productivity of algorithm invention, we would finally like to propose the following workflow:

- Given a specification;
- 1. Invent an algorithm;
- 2. Prove its correctness automatically;
- 3. In case of a successful prove $\mathcal{OK}$;
- 4. In case of a failing prove, analyze the counter-examples and correct the algorithm (or the specification);
- 5. GoTo 2.

References

1. L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2006*, Paphos, Cyprus, November 2006. To appear.
2. Begnaud Francis Hildebrand. *Introduction to Numerical Analysis: 2nd Edition.* Dover Publications, Inc., New York, NY, USA, 1987.
3. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2 edition, 1992.
4. W. J. Langford, T. A. A. Broadbent, and R. L. Goodstein. Obituary: Professor Eric Harold Neville. *The Mathematical Gazette*, 48(364):131–145, 1964.
5. E. H. Neville. Iterative Interpolation. *Journal of the Indian Mathematical Society*, 20:87–120, 1934.