A Purely Logical Approach to Program Termination

EXTENDED ABSTRACT

Mădălina Eraşcu* Tudor Jebelean
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
{merascu,tjebelea}@risc.uni-linz.ac.at

We present our work in progress concerning the logical foundations of the analysis of termination for imperative recursive programs. The analysis is based on forward symbolic execution [12] and functional semantics. The distinctive feature of our approach is the formulation of *the termination condition as an induction principle* developed from the structure of the program with respect to iterative structures (recursive calls and `while` loops). Moreover the termination condition insures the existence and the uniqueness of the function implemented by the program. Note that the existence is not automatic, because a recursive program corresponds, logically, to an implicit definition. It is interesting that this inductive termination condition can be also used for proving the uniqueness of the function as well as the total correctness of the program. We show in this paper how to prove the existence of the implemented function in the case of single recursion programs (programs with at most one recursive call on each branch). The method can be applied however to all imperative recursive programs, where recursive calls are outside the loops. For other programs, termination analysis appears to involve co-recursive functions and it is subject to further investigation. The methods presented here are under implementation in the *Theorema* system [2].

Related work. Existing static analysis methods in the Floyd-Hoare style [7, 10] for proving termination of programs with loops consist in manually annotate the loop with a termination term [9], or to synthesize the termination term automatically using various techniques mostly from linear [integer] programming [15, 1]. These approaches can be seen in the context of our work as methods for proving certain classes of such logically expressed termination conditions that we generate. A recent approach for termination of functional programs is based on the comparison of infinite paths in the control flow graph and in „size-change graphs", comparison that is reduced to the inclusion test for Büchi automata [13]. Automated tools supporting termination analysis are e.g. Terminator [3, 8], ACL2 [11], and termination tools for term rewriting systems (`http://www.termination-portal.org/`).

Theoretical background. Our approach is purely logic, meaning that the program correctness is provable in predicate logic, without using any additional theoretical model for program semantics or program execution, but only using the theories relevant to the predicates, constants and functions present in the program text. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) We call such theories Υ *object theories*. We consider two kinds of functions in the object theory: *i*) *basic* – they have only input conditions, but no output conditions (e.g. arithmetic operations in various number domains); and *ii*) *additional* – they are functions implemented by other programs or bounded arithmetical operations, and in the process of verification conditions generation only their specification will be used.

Additionally, we consider a *meta-theory* which includes the programming constructs (statements), the program itself, as well as the terms and the formulae from the object theory, which are *meta–terms* from the point of view of the meta-theory, and they behave like *quoted*. The programming statements are: abrupt statements (`break, return`), assignments – including recursive calls, conditionals and `while`

*Supported by Upper Austrian Government and Austrian Science Foundation (FWF) grant W1214–DK1

loops. The statements contain formulae and terms from the object theory. A program P is a tuple of statements and is annotated with pre- and postcondition, the logic formulae $I_f[\alpha]$ and $O_f[\alpha,\beta]$, respectively. It takes as input a certain number of variables and it returns a single value β. For simplicity we consider a single input variable (denoted conventionally by α). (Clearly this formalism can be easily extended to several input and output variables).

The meta-theory contains further constructs for reasoning about programs: The meta-predicate Π checks that a program is syntactically correct, that every branch contains a `return` statement, that `break` statement occurs only inside loops, and that each variable is initialized before it is used [1]. The meta-level function Σ creates an object-level formula containing a new [second order] symbol f denoting conventionally the function defined by the program. It generates a conjunction of formulae with the shape:

$$\underset{\alpha:I_f}{\forall} \Phi \Rightarrow (f[\alpha] = t),$$

having the following meaning: the expression for f is the symbolic term t, conditioned by the object-level formula Φ – the accumulated conditions coming from the analysis of each statement on the respective path. This formula is universally quantified over the input variable α satisfying the input condition I_f of the program. (Please note that in this paper we depart from the classical notation for application of functions and predicates, in that we use the square brackets instead of the usual round brackets.) We consider this formula as being the *semantics* of the program P in the following sense: the function f implemented by the program satisfies $\Sigma[P]$, in other words $\Sigma[P]$ is the implicit definition of the function f. Note that Σ effectively translates the original program into a *functional* program. From this point on, one could reason about the program using e.g. the Scott fixpoint theory ([14], pag. 86), however we prefer a purely logical approach. The meta-level function Γ generates two kinds of verification conditions insuring the partial correctness of the program. *Safety conditions* are formulae with the shape $\Phi \Rightarrow I_h[t]$, where I_h is the input condition of some function h called with the current symbolic value t, and the formula Φ accumulates the conditions on the respective branch. *Functional conditions* are formulae checking that the output condition on the currently returned value is a consequence of the accumulated conditions on the respective branch. Finally, the meta-level function Θ generates a termination condition for the recursive program and for each `while` loop.

The detailed formalization of the predicate and meta-functions are presented in [4, 5, 6].

Single recursion programs. We present in this paper the main meta-theorems concerning the programs whose text contains *at most one recursive call on each branch*. It is quite straightforward to show that such programs can always be expressed as in (1), where Q is a predicate and S, C, and R are functions defined using the constructs present in the program text, possibly using conditionals but no recursion.

$$P: \quad f[\alpha] = \quad \underline{\texttt{if}}\ Q[\alpha]\ \underline{\texttt{then}}\ S[\alpha]\ \underline{\texttt{else}}\ C[\alpha, f[R[\alpha]]] \tag{1}$$

The semantics formula $\Sigma[P]$ and termination condition $\Theta[P]$ for such a program are (2) and (3), respectively.

$$\underset{\alpha:I_f}{\forall} \wedge \begin{cases} Q[\alpha] \Rightarrow (f[\alpha] = S[\alpha]) \\ \neg Q[\alpha] \Rightarrow (f[\alpha] = C[\alpha, f[R[\alpha]]]) \end{cases} \tag{2}$$

$$\underset{\alpha:I_f}{\forall} \wedge \left\{ \begin{cases} Q[\alpha] \Rightarrow \pi[\alpha] \\ \neg Q[\alpha] \wedge \pi[R[\alpha]] \Rightarrow \pi[\alpha] \end{cases} \right\} \Rightarrow \underset{\alpha:I_f}{\forall} \pi[\alpha] \tag{3}$$

(We use as notations: $x : I_f$ for "x satisfying $I_f[x]$" and $\wedge$ with curly brackets for conjunction of several formulae.) Note that both formulae are at object level. Also, (2) is an implicit definition of f.

[1] The check of initialized variables is purely syntactic, thus it may force some (logically) unnecessary initializations.

In formula (3), π is a *new constant symbol*, thus in fact it behaves like a universally quantified predicate. This is why this formula is in fact an induction principle. Note also that the termination condition abstracts some details of the actual program (functions S and C), because they are in fact not important for termination.

The rationale of this formula is as follows: The left-hand side of the implications represents a property which should be fulfilled by the predicate $\pi[\alpha]$ (*"the program terminates on input α"*), property which, in case of recursive calls includes also the predicate $\pi[R[\alpha]]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f. However there may be many predicates which have this property (for instance "True"). Intuitively, we consider that the predicate expressing termination is the strongest predicate obeying this property. Since the new constant π behaves like a universally quantified (second order) variable, the formula states that the input condition I_f is stronger than any such predicate, thus it is stronger than the termination predicate. Therefore, the program terminates for any values of the input variable which fulfills the input condition.

Correctness of the method. The total correctness formula for single recursion programs is expressed as: "The formula $\underset{\alpha}{\forall} I_f[\alpha] \Rightarrow O_f[\alpha, f[\alpha]]$ is a logical consequence of the semantics $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ is contradictory to the theory, which may happen when the program is recursive. Therefore, one proves first that the existence (and the uniqueness) of a f satisfying $\Sigma[P]$ is a logical consequence of the verification conditions. This follows from the termination condition.

We give now the main steps of the development leading to this fact. Please note that we use n, m as natural numbers, and n^+ for the successor function.

Lemma 1. *(Existence of the repetition function.) The formula*

$$\underset{h}{\forall}\,\underset{G}{\exists}\,\underset{x}{\forall}\left(G[0,x] = x \wedge \underset{n:\mathbb{N}}{\forall}\left(G[n^+,x] = h[G[n,x]]\right)\right)$$

is a logical consequence of the natural number theory.

Proof. Let x be arbitrary but fixed.

One proves first $\underset{m:\mathbb{N}}{\forall}\,\underset{H}{\exists}\left(H[0] = x \wedge \underset{n<m}{\forall} H[n^+] = h[H[n]]\right)$ by natural induction on m. From here by Skolemization on H one obtains $\underset{\mathscr{H}}{\exists}\,\underset{m:\mathbb{N}}{\forall}\left(\mathscr{H}[m][0] = x \wedge \underset{n<m}{\forall} \mathscr{H}[m][n^+] = h[\mathscr{H}[m][n]]\right)$. Furthermore one can prove $\underset{n:\mathbb{N}}{\forall}\,\underset{m\geq n}{\forall} \mathscr{H}[m][n] = \mathscr{H}[n][n]$ by natural induction on n and by taking $g[n] = \mathscr{H}[n][n]$ one has (since x was arbitrary) $\underset{x}{\forall}\,\underset{g}{\exists}\left(g[0] = x \wedge \underset{n:\mathbb{N}}{\forall} g[n^+] = h[g[n]]\right)$ which by Skolemization on g gives the desired formula (with notation $G[n,x]$ instead of $G[x][n]$). □

Remark 1. The function $G[n,x]$ is usually denoted as $h^n[x]$.
Remark 2. It is straightforward to show that $h^n[h[x]] = h^{n^+}[x]$.

The subsequent properties need the theory of natural numbers, although we do not specify this explicitly.

Lemma 2. *(Existence of the recursion index.) The formula* $\underset{\alpha:I_f}{\forall}\,\underset{n:\mathbb{N}}{\exists} Q[R^n[\alpha]]$ *is a logical consequence of the termination condition* (3) *and the safety verification conditions.*

Proof. The proof uses the induction principle given in (3), where $\pi[\alpha]$ is $\underset{n:\mathbb{N}}{\exists} Q[R^n[\alpha]]$. One needs to use the safety conditions and the property of h^n given above. □

Remark 1. One can define now a function (the recursion index of α) $M[\alpha] = \min\{n \mid Q[R^n[\alpha]\}$ because the set is nonempty.
Remark 2. It is straightforward to show that $M[R[\alpha]]^+ = M[\alpha]$.

Theorem 1. *(Existence of the function implemented by the program.) The formula* (2) *is a logical consequence of the termination condition* (3) *and the safety verification conditions.*

Proof. The proof is similar to the one from *Lemma 1*, only that instead of the running argument n we use α with a certain recursion index.

One proves first:

$$\underset{m:\mathbb{N}}{\forall}\ \underset{F}{\exists}\ \underset{\alpha:I_f}{\forall}\ (M[\alpha] \le m) \Rightarrow \big(\big(Q[\alpha] \Rightarrow F[\alpha] = S[\alpha]\big) \wedge \big(\neg Q[\alpha] \Rightarrow F[\alpha] = C[\alpha, F[R[\alpha]]]\big)\big) \tag{4}$$

by natural induction on m.

By Skolemizing F from (4) one obtains:

$$\underset{\mathscr{F}}{\exists}\ \underset{m:\mathbb{N}}{\forall}\ \underset{\alpha:I_f}{\forall}\ (M[\alpha] \le m) \Rightarrow \big((Q[\alpha] \Rightarrow \mathscr{F}[m][\alpha] = S[\alpha]) \wedge (\neg Q[\alpha] \Rightarrow \mathscr{F}[m][\alpha] = C[\alpha, \mathscr{F}[m][R[\alpha]]])\big)$$

Furthermore one can prove $\underset{\alpha:I_f}{\forall}\ \underset{m:\mathbb{N}}{\forall}(m \ge M[\alpha]) \Rightarrow (\mathscr{F}[m][\alpha] = \mathscr{F}[M[\alpha]][\alpha])$ by the induction given in the formula (3) (taking as $\pi[\alpha]$ the formula above without the quantifier for α).

Finally one takes $f[\alpha] = \mathscr{F}[M[\alpha]][\alpha]$. □

Remark. Uniqueness of f is straightforward: take f_1, f_2 satisfying (2) and use (3) with $\pi[\alpha]$ as $f_1[\alpha] = f_2[\alpha]$.

Theorem 2. *(Total correctness.) The formula* $\underset{\alpha}{\forall} I_f[\alpha] \Rightarrow O_f[\alpha, f[\alpha]]$ *is a logical consequence of the program semantics and the verification conditions.*

Proof. The proof is straightforward by taking in (3) $\pi[\alpha]$ as $O_f[\alpha, f[\alpha]]$. This is because the left-hand side of the (3) becomes identical to the functional conditions generated for partial correctness. □

Programs containing `while` loops. Our approach can be easily extended to programs containing arbitrarily nested possibly abrupt terminating `while` loops[2], by transforming the loop body B into a tail recursive function where the loop invariant ι represents the specification of the new function. The parameters of the virtual function are the so called *critical variables* – the variables which are modified within the loop body. The actual arguments of the call are the values of the critical variables when entering the loop. The template semantics and termination formulae are similar to those corresponding to primitive recursive functions, namely if the `while` loop is (5) then the semantics is (6) and the termination condition is (7).

$$W: \quad \underline{\texttt{while}}\ \varphi[\delta]\ \ \underline{\texttt{do}}\ \ \iota, B \tag{5}$$

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} \neg\varphi[\delta] \Rightarrow (f[\delta] = \delta) \\ \varphi[\delta] \Rightarrow (f[\delta] = f[R[\delta]]) \end{cases} \tag{6}$$

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{cases} \neg\varphi[\delta] \Rightarrow \pi[\delta] \\ \varphi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{cases} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \tag{7}$$

In case of nested loops, the operations from the body of the inner loops are not reflected explicitly in the clauses of the semantics and verification conditions, except if a `return` – at all levels or a `break`–

[2]Currently we do not treat the situation where the loop body contains a recursive call to the main function, because this will lead to mutual recursion.

in non-nested loops is encountered. It is the loop invariant that encodes them and that it is used in the further analysis of the program. The details of the formalization are found in [6]. The correctness proof proceeds similar to the case of primitive recursive functions.

Conclusions and future work. During the construction of a mathematical theory, when we define a new function in an implicit way, we prove the existence of it in order to avoid introducing a contradiction in our theory. Likewise, during the construction of a software system, when we program a new function, we prove the termination of it, in order to insure the effectiveness of our system. The work presented here is a step towards showing that these two situations essentially reduce to the same logical operations.

Further work includes the investigation of termination theory for programs with multiple recursion and with nested recursion, as well as the development of methods for proving the verification conditions by combining logical and algebraic algorithms.

References

[1] A. Bradley, Z. Manna, and H. Sipma, *Linear Ranking with Reachability*, Proc. 17^{th} Intl. Conference on Computer Aided Verification (K. Etessami and S. Rajamani, eds.), Lecture Notes in Computer Science, vol. 3576, Springer Verlag, July 2005.

[2] 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*, Journal of Applied Logic **4** (2006), no. 4, 470–504.

[3] B. Cook, A. Podelski, and A. Rybalchenko, *CFL-Termination*, Tech. Report MSR-TR-2008-160, Microsoft Research, 2008.

[4] M. Eraşcu and T. Jebelean, *Practical Program Verification by Forward Symbolic Execution: Correctness and Examples*, Austrian-Japan Workshop on Symbolic Computation in Software Science (B. Buchberger, T. Ida, and T. Kutsia, eds.), 2008, pp. 47–56.

[5] M. Eraşcu and T. Jebelean, *A Calculus for Imperative Programs: Formalization and Implementation*, Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (S. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, and D. Zaharie, eds.), IEEE, 2009, pp. 77– 84.

[6] M. Eraşcu and T. Jebelean, *A Purely Logical Approach to Imperative Program Verification*, Tech. Report 10-07, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2010.

[7] R. Floyd, *Assigning Meaning to Programs*, Proc. of Symposia in Appl. Math. American Mathematical Society, 1967.

[8] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, *Proving that Non-blocking Algorithms don't Block*, POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 2009, pp. 16–28.

[9] D. Gries, *The Science of Programming*, Springer, 1981.

[10] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM **12** (1969), no. 10, 576–580.

[11] M. Kaufmann and J. S. Moore, *An Industrial Strength Theorem Prover for a Logic Based on Common Lisp*, Software Engineering **23** (1997), no. 4, 203–213.

[12] J. King, *Symbolic Execution and Program Testing*, Communications of the ACM **19** (1976), no. 7, 385–394.

[13] C. S. Lee, N. Jones, and A. Ben-Amram, *The Size-Change Principle for Program Termination*, SIGPLAN Not. **36** (2001), no. 3, 81–92.

[14] J. Loeckx, K. Sieber, and R. Stansifer, *The Foundations of Program Verification*, John Wiley & Sons, Inc., New York, NY, USA, 1984.

[15] A. Podelski and A. Rybalchenko, *A Complete Method for the Synthesis of Linear Ranking Functions*, VMCAI, 2004, pp. 239–251.