A Purely Logical Approach to the Termination of Imperative Loops

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

***Abstract*—We present and illustrate a method for the generation of the termination conditions for nested loops with abrupt termination statements. The conditions are (first-order) formulae obtained by certain transformations of the program text. The loops are treated similarly to calls of recursively defined functions. The program text is analyzed on all possible execution paths by forward symbolic execution using certain meta-level functions which define the syntax, the semantics, the verification conditions for the partial correctness, and the termination conditions. The termination conditions are expressed as induction principles, however, still in first-order logic.**

Our approach is simpler than others because we use neither an additional model for program execution, nor a fixpoint theory for the definition of program semantics. Because the meta-level functions are fully formalized in predicate logic, it is possible to prove in a purely logical way and at object level that the verification conditions are necessary and sufficient for the existence and uniqueness of the function implemented by the program.

***Index Terms*—program analysis and verification, symbolic execution, semantics, induction, termination, *Theorema* system**

I. Introduction

We present the theoretical foundations of a formal method for handling the total correctness of arbitrary nested (possibly abruptly terminating) `while` loops in imperative programs. The goal of our approach is mainly foundational: *we aim at the identification of the minimal logical apparatus necessary for formulating and proving (in a computer assisted manner) a correct collection of methods for program verification.* The study of such a minimal logical apparatus should increase the confidence in program verification tools and possibly reveal some foundational relations between logic and programming. Moreover this computer aided formalization may open the possibility of reflection of the method on itself (treatment of the meta-functions as programs whose correctness can be studied by the same method).

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 `while` loops. This termination condition ensures the logical existence of the function implemented by the loop. (The existence is not automatic, because a loop corresponds from the logical point of view to an implicit definition.) Moreover, the termination condition can be also used for proving the uniqueness of the function as well as the total correctness of the loop.

Syntactically, a program is considered to be a tuple of statements, which contains terms and formulae expressed in an *object theory* expressing the properties of the objects manipulated by the program. (By a theory we understand a set of formulae in the language of predicate logic.) Therefore, the program statements are meta-constructs which incorporate object terms and object formulae as quoted meta-terms. Currently, we consider programs which have some input parameters and produce a single return value, thus a program defines a function.

The *semantics* of the program is the object level formula which defines logically the function implemented by the program. This formula is constructed by traditional forward symbolic execution: the program is executed on symbolic arguments, and a clause for the implicit definition of the program function is generated on each execution path of the program. We consider the conjunction of these clauses as being the semantics of the program. In a similar way, on each path, one generates the *partial correctness conditions* as safety, functional, and assertive conditions. Separately, the *termination condition* is generated as an induction principle. Remarkably, all the conditions are expressed in the object logic.

This paper is an extension of [9], where we introduced an algorithmic approach for checking the syntax and generating the semantics and the verification conditions (for partial correctness and for termination) for imperative programs having *non-nested, normal terminating,* `while` *loops*, using the principles presented above. This was achieved, basically, by constructing functional meta-definitions for the notions of syntax, semantics, partial correctness and termination. The semantics of a loop was captured by its invariant.

This paper enhances [9] in the following aspects:

- the formal meta-definitions (full description in [10] – this paper emphasizes the termination in Section IV) handle now imperative recursive programs with *arbitrary nested* `while` *loops*, possibly *abrupt terminating via* `return` *and* `break`, where the recursive calls can occur only outside loops,
- the loop semantics is expressed as an implicit definition of a [tail-] recursive function, and

- the correctness of method for this class of programs is presented (Section V), where the crucial role is played by the existence of semantics function, and the inductive termination condition.

The method is completely formalized as functional meta-definitions of the syntax checker, of the semantics constructor and of the generators of the verification conditions. These meta-level functions take the program text, the specification (input and output conditions) and, eventually, assertions like the loop invariant or others introduced by the `Assert` statement. The approach is also universal because the programming language itself is predicate logic (terms and formulae from the object theory), except for the few meta-constructs which represent the basic imperative statements (assignment, conditionals, loops, abrupt statements: `break, return`), and for `Assert` construct.

A `while` loop is treated similarly to a recursive call of a new function together with the definition and specification of it. The semantics of a loop is a logical implicit definition of a function. The invariant has the role of input and output condition of this virtual 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 method is implemented in Mathematica [24] on top of the *Theorema* system [5]. Because the formalization of the functional meta-definitions are given in the ,,pattern matching" style, that is, it exhibits the behavior of the meta-function for various specific classes of arguments, the differences between the definitions presented in [10] and the real implementation are minor.

Our approach is presented in full detail in [10]; here we emphasize the termination of loops. In particular, we present the proof of the fact that the termination condition ensures the existence and the uniqueness of the function implemented by the loop. The proof is elementary, using only natural number induction and few inference rules.

Related Work

Our approach follows the principles of forward symbolic execution [15] and functional semantics [19], but additionally gives formal definitions in a meta-theory for the meta-level functions which define the syntax, the semantics, and the verification conditions. To our knowledge there is no other work on symbolic execution approaching the verification problem in a fully formal way.

However, the ideas from the formalization of the calculus are not completely new; [17] describes the behavior of concurrent systems as relation between the variables in the current state and in the post-state. A similar approach is encountered in [2] where the program equations (involving relation between current and post-state) are used to express nondeterminacy and termination. In the same manner, [22] presents the formal calculus for imperative languages containing complex structures. Specification languages used in the framework of verification tools also use this concept – see e.g JML [6].

The most well-known techniques for proving that a loop terminates is to manually annotate it with a termination term [13], to synthesize the termination term based on the loop behavior [3], [4], [21] or to compute the closure of some well-founded relations [7]. They can be seen in the context of our work as methods for proving certain classes of inductive termination conditions that we generate.

The interactive theorem prover Coq [1] mechanizes the most well-known semantics for imperative languages (operational, denotational and axiomatic). Our approach is in the spirit of the axiomatic semantics, if we think to the fact that we annotate the program (input and output specification, invariants for loops and `Assert` commands). But actually we transform the imperative program into a functional one (a tail-recursive function for each loop), thus one may expect a relation to operational (call-by-value) or denotational semantics (based on fix-point theory) for functional languages. On one side, we are not interested how the functions evaluate their arguments, therefore we do not use operational semantics. On the other side, denotational semantics based on fix-point theory needs an additional model for dealing with nonterminating programs. Our approach uses implicitly the semantics of predicate logic.

Most of the proof assistants provide infrastructure for proving/disproving the termination of classical examples with general recursion. ACL2 [14] handles total functions that must be proved total at the definition time; sometimes the system is able to infer this fact. Isabelle [20], HOL4 [12] and Coq [1] are basically using the recursion package TFL [23] and thus allow definitions of total recursive functions by using the fixed-point operators and well-founded relations supplied by the user. Proving termination reduces to show that the relation is well-founded and the arguments of the recursive calls are decreasing. Our approach is equivalent, in the sense that the termination condition is equivalent to the well-foundedness of the partial order defined by the transformation of the critical variables within the loop.

The treatment of termination in [16] also uses inductive conditions extracted from the program recursions, but in the form of implicit definitions of domains (set theory is also needed). However, the existence of such inductively defined objects is not proved directly.

Since our study is foundational, it constitutes a complement and not a competitor for practical work dealing with termination proofs, like e. g. termination of term rewriting systems http://www.termination-portal.org/, the size-change termination principle [18] or the approaches for proving the termination of industrial-size code (Microsoft Windows Operating System Drivers) [7].

II. Logical Foundations of Imperative Loops

Our approach is purely logic, meaning that the program correctness, and implicitly loop correctness, are expressible 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. We call such theories *object theories*.

A *meta-theory* (in predicate logic with equality) is further constructed for reasoning about programs. The meta-theory contains specific functions and predicates from the set theory. Moreover it needs the tuple theory: $\langle ... \rangle$ denotes a tuple, $\frown$ is the concatenation of tuples; $x \to t$ is a pair denoting the replacement of the variable x with the term t (a set of replacements is a substitution, and substitutions can also be composed); as well as appropriate function symbols for the construction of program statements (assignment, conditionals, loops, abrupt statements: `break, return`).

The program statements and the program itself are meta-terms. Also the terms and the formulae from the object theory are *meta-terms* from the point of view of the meta-theory, and they are considered *quoted* (because the meta-theory does not contain any equalities between programming constructs, and also does not include the object theory).

The expressions composing the definitions of the meta-level predicate and functions presented below are to be understood as universally quantified over the meta-variables of various types: $v \in V \subset \mathcal{V}$ is an initialized variable, $t \in \mathcal{T}$ is a term, φ is a boolean expression, B, P_T and P_F are tuples of statements representing the loop body and the two paths corresponding to the `if` statement, respectively. ι denotes conventionally the loop invariant which holds at the beginning of the loop and is inductively preserved by each iteration of the loop. The user should provide such an invariant. We denote conventionally by δ the critical variables. For simplicity we consider a single critical variable in loops which we denote as δ. It is straightforward to extend this formalism to tuples of critical variables.

A program P is a tuple of statements and is documented with pre- and postconditions. It takes as input a certain number of variables and it returns a single value conventionally denoted β.

The meta-theory also contains the properties of the meta-predicate Π (syntax checker) and meta-functions Σ, Σ', and Σ'' (semantics generators), Γ (verification conditions generator), and Θ, Θ', and Θ'' (termination condition generators).

All meta-functions use forward symbolic execution: the state of the program is represented by a formula Φ and by a substitution σ. The formula contains the accumulated condition on the current execution path (the path condition). The substitution assigns the current symbolic values (terms depending on the input) to the currently initialized variables. Program analysis proceeds as follows:

- Initially new symbolic constants are assigned to the input variables.
- After that, the program analysis proceeds in forward manner, statement by statement.
- An assignment updates the substitution σ using the variable and the term of the assignment.
- A conditional (`if`) splits the analysis of the corresponding execution path and adds the condition formula and its negation to the two new path conditions.
- `return` ends the analysis on the current execution path and `break` ends the analysis of the current loop.

A `while` statement splits the analysis of the program in three paths adding new path constraints to Φ as follows:

1) One path analyzes the statements occurring after the loop – the case when the loop is not executed at all.
2) On the second path the loop is executed symbolically using fresh values for the critical variables, which are assumed to fulfill the invariant and the loop condition.
3) The third path continues the analysis of the program after the loop, by assigning fresh values to the critical variables, which are assumed to fulfill the invariant and the negated loop condition. Exception make the critical variables of abruptly terminating loops via `break`, which, in our approach, fulfill the invariant and an assertion specified by the user.

At the end of the analysis, the original constant symbols are translated back to the input variables.

Remark 1: In the enumeration above, 1) does not subsume 3): in 1) the variables have exact values, while in 3) they have symbolic values. Moreover, for reasoning about loops, in particular about their correctness, one needs only the item 2) in the enumeration above.

The output of the meta-functions consists in a tuple of formulae, each of them to be understood as universally quantified over the free variables.

We illustrate the method using *Example 1* (*Search in a bidimensional array*) containing two nested, abrupt terminating loops. Line 1 and 2 represent the input, respectively the output condition of the program. The loops are annotated with the invariants ι_1 and ι_2.

III. Syntax, Semantics, and Partial Correctness

We first introduce a meta-predicate Π which checks the syntax and a meta-function Σ which constructs the semantics. These are not actually needed for the implementation of a program verification system. They are only needed in order to reason about the effect of the verification condition generator. For instance, all statements about the effect of the meta-functions can be formulated only on programs which fulfill the predicate Π. Likewise, the effect of a program P is expressed as a logical formula $\Sigma[P]$, which constitutes the implicit definition of the function realized by the program. Additionally, we generate the semantics of each loop as an implicit definition of the function implemented by the loop on the critical variables.

The verification conditions for partial correctness are generated by a meta-function Γ.

Since the focus of this paper is on the termination, we do not give here the definitions of Π, Σ and Γ (see details in [10]), but only the main description.

A. Syntax

The predicate Π checks a program for syntactic correctness, including the fact that each variable is initialized, that each

Algorithm 1 Search in a bidimensional array

1. in A: array of reals; m, n: integers; e: real where $m > 0$, $n > 0$
2. out β: integer or tuple where
$(\underset{0\le k<m}{\exists}\ \underset{0\le l<n}{\exists} A_{\langle k,l\rangle} = e \Rightarrow A_\beta = e) \wedge$
$(\underset{0\le k<m}{\forall}\ \underset{0\le l<n}{\forall} A_{\langle k,l\rangle} \neq e \Rightarrow (\beta = -1))$
3. local int i, j, s;
4. $s := 0; i := 0$;
5. while $(i < m)$
6. $\quad \iota_1 \colon 0 \le i \le m \wedge \underset{0\le k<i}{\forall}\ \underset{0\le j<n}{\forall} A_{\langle k,j\rangle} \neq e$,
7. $\quad j := 0$;
8. $\quad$ while $(j < n)$
9. $\quad\quad \iota_2 \colon 0 \le j \le n \wedge \underset{0\le i<m}{\forall}\ \underset{0\le l<j}{\forall} A_{\langle i,l\rangle} \neq e$,
10. $\quad\quad$ if $(e = A_{\langle i,j\rangle})$
11. $\quad\quad\quad s := 1$;
12. $\quad\quad\quad$ break;
13. $\quad\quad j := j + 1$;
14. $\quad$ Assert$[\iota_2 \vee (s = 1 \ \wedge e = A_{\langle i,j\rangle})]$;
15. $\quad$ if $(s = 1)$
16. $\quad\quad$ return$[\langle i, j\rangle]$;
17. $\quad i := i + 1$;
18. return[-1]

execution path contains a return statement, and that the break statement occurs only in while loops. Moreover, we require that each loop terminating abruptly via break is annotated with an assertion, other than the invariant. The reason is that one can not characterize precisely the critical variables at the exit point of the loop as fulfilling the invariant and the negated loop condition like in the case of normal terminating loops. Therefore, the critical variables from loops terminating abruptly via break are known to fulfill the invariant and the assertion. This assertion is further used in the analysis of the program after the loop and replaces the negated loop condition which is known to be fulfilled only at the end of normal terminating loops.

For instance our illustrating example is syntactically correct.

B. Semantics

For programs containing while loops there are two possibilities for constructing their semantics. The one presented in [9] uses the loop condition, the invariant and the values computed by the loop for the critical variables in order to characterize the loop. Although simple, this technique considers the loop as a black box: its effect is encoded into the invariant and the operations inside the loop body are not taken into consideration. We express in this paper a loop as a recursive function (1) because this view allows us to prove the correctness of our method in Section V, similarly to the single recursion programs [11].

We define the semantics of a loop via the meta-level function Σ as being an implicit definition *at object level* of the function implemented by the loop. Each formula on the left hand side of (1) is a conditional definition for $f[\delta]$ and depends on the accumulated [negated] conditions φ coming from the conditionals and from the loop constructs (invariants and loop conditions). The value of $f[\delta]$ is the symbolic value of the returned term according to the current substitution σ on the respective non-recursive program paths and, respectively, the value of f with the actual arguments (saved in σ_W) of the recursive call, for the iterative paths. Note that the ellipsis in (1) might represent other conditional definitions for f coming from other iterative paths, and from the analysis of the abrupt terminating statements.

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} \neg\varphi \Rightarrow (f[\delta] = \delta) \\ \varphi \Rightarrow (f[\delta] = f[\delta\sigma_W]) \\ \ldots \end{cases} \tag{1}$$

For instance, the semantics of the loops in our example is given below, with certain notational conventions: f_1 and f_2 are the symbols standing for the functions implemented by the outer, respectively the inner loop, while the tuple of numbers after each formula represents the corresponding execution path.

Semantics of the outer loop.

$$\underset{i,j,s:\iota_1}{\forall} \wedge \begin{cases} i \ge m \Rightarrow (f_1[i,j,s] = \langle i,j,s\rangle) \\ \quad \langle 5,6\rangle \\ i < m \wedge 0 \ge n \wedge \big((0 \le n \wedge \underset{0\le i<m}{\forall}\ \underset{0\le l<0}{\forall} A_{\langle i,l\rangle} \neq e) \\ \quad\quad \vee (s = 1 \wedge A_{\langle i,0\rangle} = e)\big) \wedge s = 1 \\ \quad\quad \Rightarrow f_1[i,j,s] = \langle i,j\rangle \\ \quad \langle 5,6,7,8,14,15,16\rangle \\ i < m \wedge 0 \ge n \wedge \big((0 \le n \wedge \underset{0\le i<m}{\forall}\ \underset{0\le l<0}{\forall} A_{\langle i,l\rangle} \neq e) \\ \quad\quad \vee (s = 1 \wedge A_{\langle i,0\rangle} = e)\big) \wedge s \neq 1 \\ \quad\quad \Rightarrow f_1[i,j,s] = f_1[i+1,j,s] \\ \quad \langle 5,6,7,8,14,15,17\rangle \\ i < m \wedge j \ge n \wedge \iota_2 \wedge \big((0 \le j \le n \\ \quad\quad \wedge \underset{0\le i<m}{\forall}\ \underset{0\le l<j}{\forall} A_{\langle i,l\rangle} \neq e) \vee (s = 1 \wedge A_{\langle i,j\rangle} = e)\big) \\ \quad\quad \wedge (s = 1) \Rightarrow f_1[i,j,s] = \langle i,j\rangle \\ \quad \langle 5,6,8,9,14,15,17\rangle \\ i < m \wedge j \ge n \wedge \iota_2 \wedge \big((0 \le j \le n \\ \quad\quad \wedge \underset{0\le i<m}{\forall}\ \underset{0\le l<j}{\forall} A_{\langle i,l\rangle} \neq e) \vee (s = 1 \wedge A_{\langle i,j\rangle} = e)\big) \\ \quad\quad \wedge s \neq 1 \Rightarrow f_1[i,j,s] = f_1[i+1,j,s] \\ \quad \langle 5,6,8,9,14,15,16\rangle \end{cases}$$

Semantics of the inner loop.

$$\underset{j,s:\iota_2}{\forall} \wedge \begin{cases} j \ge n \Rightarrow (f_2[j] = j) \\ \quad \langle 8,9\rangle \\ j < n \wedge (e = A_{\langle i,j\rangle}) \Rightarrow (f_2[j,s] = \langle i,j\rangle) \\ \quad \langle 8,9,10,11,12\rangle \\ j < n \wedge (e \neq A_{\langle i,j\rangle}) \Rightarrow (f_2[j,s] = f_2[j+1,s]) \\ \quad \langle 8,9,10,13\rangle \end{cases}$$

C. Partial Correctness

The meta-function Γ generates the partial correctness conditions. These ensure *safety* (all functions are used with appropriate values of the arguments) and *functional correctness* (the values returned by the main function satisfy the output specification). Additional verification conditions, called *assertive*

conditions, are the conditions whose goal is represented by an intermediary program assertion introduced by the `Assert` construct. Altogether, they ensure the partial correctness. An example is the formula:

$m > 0 \wedge n > 0 \wedge 0 \geq m \Rightarrow (\underset{0\leq k<m}{\exists}\ \underset{0\leq l<n}{\exists} A_{\langle k,l\rangle} = e \Rightarrow (A_\beta = e)) \wedge (\underset{0\leq k<m}{\forall}\ \underset{0\leq l<n}{\forall} A_{\langle k,l\rangle} \neq e \Rightarrow (-1 = -1))$,

obtained from the analysis of the path: $\langle 4, 5, 18\rangle$, which corresponds to the case when the loop is not executed.

In the case of loops, the safety, functional, and assertive conditions are generated as for recursive programs. Additionally to these, verification conditions ensuring the invariant at first loop entry, as well as the preservation of the invariant at each iteration (including the exit values in case of `break`) are generated.

IV. Termination

Non-recursive programs containing nested abrupt terminating `while` loops terminate if each loop terminates. We approach the termination by considering the loop as separate module (program or loop) whose termination condition is:

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} \neg\varphi \Rightarrow \pi[\delta] \\ \varphi \wedge \Phi \wedge \pi[\delta\sigma_W] \Rightarrow \pi[\delta] \\ \ldots \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \qquad (2)$$

In formula (2), 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. The formula consists of an implication between two universally quantified parts, both over δ (standing for the loop critical variable[s]) which satisfy the loop invariant ι. The left-hand side is a conjunction of implicational clauses. The first clause corresponds to the end of the loop (φ is the loop condition), and the other clauses correspond to the paths of execution of the loop. Each path has associated a path condition Φ and a substitution σ_W which encodes symbolically the effect of this path on the critical variable[s] δ.

We now explain the rationale behind (2). Let us consider the predicate $\tau[\delta]$: "the loop terminates on input δ" (whose definition we do not actually know). The left-hand side of the implication represents a property $T[\pi]$ which should be fulfilled by this predicate τ. Intuitively, this property states that the loop terminates if the condition φ is not fulfilled, and furthermore, corresponding to each execution path, it states that the loop terminates on δ if it terminates on the values updated by the execution of the loop $\delta\sigma_W$. Intuitively, we consider that the predicate expressing termination is the strongest predicate obeying this property T. The termination condition states that the invariant ι is stronger than any predicate fulfilling T – thus it will be also stronger than τ. In this way we can express termination without explicit use of τ.

Therefore, the condition states that the loop terminates for any values of the critical variable which fulfills the invariant, in particular for the values of the critical variable at the entry point of the loop because they preserve the invariant. This is, however, only an intuitive explanation, and in the next section we show rigourously that the termination condition is sufficient for the existence and uniqueness of the function implemented by the loop.

Loop analysis implies collecting in Φ the conditions of the `if` statements, the invariants of the inner loops and the output characterization of the additional functions encountered – if it is the case, and also formulae of the type $\pi[\delta\sigma_W]$. The operations in the inner loops do not appear explicitly in the outer ones (they are encoded in the loop invariant), except for `break` – if the currently analyzed module has non-nested loops, and for `return` – at any level.

As in the semantics formula (1), the ellipsis in (2) might represent other conditional definitions for f coming from other iterative paths, and from the analysis of the abrupt terminating statements.

The termination condition, one for each loop, is generated automatically by the meta-level functions Θ, Θ' and Θ''.

The meta-level function Θ analyzes the current module and specializes itself into Θ' and, respectively, into Θ'' for modules which contain nested loops, because the `break` statement has different behavior for non-nested, respectively, for nested loops. On one hand, for non-nested loops, both `return` and `break` terminate the loop (Definitions 2.1, 2.2). On the other hand, in nested loops, `return` terminates all loops and the program itself (Definition 3.1) and `break` terminates only the innermost loop that contains it (Definition 3.2).

The main function Θ depends on the global variable representing the initial program state. The truth constant $\mathbb{T}$ is used as assumption at the beginning of the program analysis (Definition 1.1) and also each time a new module is analyzed (Definition 1.7.2). This assumption is sufficient because we generate termination conditions only for loops specified by the corresponding invariant. The meta-level function works by symbolic execution, generating a list of termination conditions, one for each loop of the program as follows: updates the program state in case of assignments (Definition 1.4), forks the analysis of the program in two execution paths in case of conditional `if` (Definition 1.5), returns the empty tuple in case of `return` statement (because termination of a module corresponds to the termination of the inner iterative structures, if they exist). Definitions 1.3 – handling `break` statements and 1.6 – handling end of `while` are applied only if the module analyzed does not contain nested loops, these cases occurring by the application of Definition 1.7.2. Each time a loop is analyzed, a new symbol π standing for an arbitrary predicate is generated and the analysis proceeds as follows: a termination condition for the currently analyzed loop is generated (Definition 1.7.1) where an auxiliary function Θ' is used, a path analyzes the loop body similarly to the program (Definition 1.7.2) and the last one continues with the analysis of the statements after the loop (Definition 1.7.3). Note that the same analysis is performed for each loop, independently of the degree of nestedness, due to Definition 1.7.2.

The inductive definitions corresponding to the meta-function Θ are as follows:

Definition 1:

1) $\Theta[P] = \Theta[\{\alpha \to \alpha_0\}, \mathbb{T}, P]\{\alpha_0 \to \alpha\}$
2) $\Theta[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P] = \langle\rangle$
3) $\Theta[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P] = \langle\rangle$
4) $\Theta[\sigma, \Phi, \langle v := t\rangle \smile P] = \Theta[\sigma\{v \to t\sigma\}, \Phi, P]$
5) $\Theta[\sigma, \Phi, \langle \underline{\texttt{if}}(\varphi) P_T, P_F\rangle \smile P] =$
$\quad \smile \begin{cases} \Theta[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P] \\ \Theta[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P] \end{cases}$
6) $\Theta[\sigma, \Phi, \langle\rangle] = \langle\rangle$
7) $\Theta[\sigma, \Phi, \langle \underline{\texttt{while}}\ (\varphi)\ \underline{\texttt{do}}\ \iota, B \smile P] =$
$\quad \smile \begin{cases} \left\langle \underset{\delta:\iota}{\forall} \wedge \left\{ \begin{matrix} (\neg\varphi\sigma_0 \Rightarrow \pi[\delta])\{\delta_0 \to \delta\} \\ \Theta'[\sigma_0, \varphi\sigma_0, B, \pi] \Rightarrow \pi[\delta] \end{matrix} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \right\rangle \quad (1) \\ \Theta[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B] \quad (2) \\ \Theta[\sigma_0, \mathbb{T}, P] \quad (3) \end{cases}$

The auxiliary functions Θ' and Θ'' behave similarly to Θ, except that they generate a disjunction of formulae (for the simplicity of the approach), one for each path analyzed, from which the termination of the loop must follow (Definition 1.7.1) and that:

- `return` statement has the same behavior in non- and nested loops: they return the accumulated path conditions (Definitions 2.1 and 3.1);
- `break` statement behaves similarly to `return` in non-nested loops (Definition 2.2), but for programs with nested loops the analysis performed in inner loops is not visible in the wrapper ones (Definitions 3.2).
- at the end of the non-nested loop, a path condition involving the termination predicate π is constructed (Definition 2.5), while the analysis performed in the nested loops is not visible in the outer loops (Definition 3.5)
- nested loops are always analyzed by the meta-function Θ'' (Definition 2.6).

Definition 2:

1) $\Theta'[\sigma, \Phi, \langle \underline{\texttt{return}}[\delta]\rangle \smile P, \pi] = \Phi\{\delta_0 \to \delta\}$
2) $\Theta'[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, \pi] = \Phi\{\delta_0 \to \delta\}$
3) $\Theta'[\sigma, \Phi, \langle v := t\rangle \smile P, \pi] = \Theta'[\sigma\{v \to t\sigma\}, \Phi, P, \pi]$
4) $\Theta'[\sigma, \Phi, \langle \underline{\texttt{if}}\ (\varphi)\ P_T, P_F\rangle \smile P, \pi] =$
$\quad \vee \begin{cases} \Theta'[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \pi] \\ \Theta'[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \pi] \end{cases}$
5) $\Theta'[\sigma, \Phi, \langle\rangle, \pi] = (\Phi \wedge \pi[\delta\sigma])\{\delta_0 \to \delta\}$
6) $\Theta'[\sigma, \Phi, \langle \underline{\texttt{while}}\ (\varphi)\ \underline{\texttt{do}}\ \iota, B\rangle \smile P, \pi] =$
$\quad \Theta''[\sigma, \Phi, \langle \underline{\texttt{while}}\ (\varphi)\ \underline{\texttt{do}}\ \iota, B]\rangle \smile P, \pi]$

Definition 3:

1) $\Theta''[\sigma, \Phi, \langle \underline{\texttt{return}}[\delta]\rangle \smile P, \pi] = \Phi\{\delta_0 \to \delta\}$
2) $\Theta''[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, \pi] = \mathbb{F}$
3) $\Theta''[\sigma, \Phi, \langle v := t\rangle \smile P, \pi] = \Theta''[\sigma\{v \to t\sigma\}, \Phi, P, \pi]$
4) $\Theta''[\sigma, \Phi, \langle \underline{\texttt{if}}\ (\varphi)\ P_T, P_F\rangle \smile P, \pi] =$
$\quad \vee \begin{cases} \Theta''[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \pi] \\ \Theta''[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \pi] \end{cases}$
5) $\Theta''[\sigma, \Phi, \langle\rangle, \pi] = \mathbb{F}$
6) $\Theta''[\sigma, \Phi, \langle \underline{\texttt{while}}\ (\varphi)\ \underline{\texttt{do}}\ \iota, B\rangle \smile P, \pi] =$
$\quad \vee \begin{cases} \Theta'[\sigma, \Phi \wedge \neg\varphi\sigma, P, \pi] \\ \Theta''[\sigma, \varphi\sigma \wedge \iota\sigma, B, \pi] \\ \Theta'[\sigma, \neg\varphi\sigma \wedge \iota\sigma, P, \pi] \end{cases}$

The termination conditions for the loops in *Example 1* are as below. There are actually two induction principles, developed from the structure of the loops. The program terminates if both loops terminates, i.e. the following two formulae hold. The tuples of numbers between the formulae represent the program lines analyzed leading to the respective path condition.

Semantics of the outer loop.

$$\underset{i,j,s:\iota_1}{\forall} \wedge \begin{cases} i \geq m \Rightarrow \pi_1[i,j,s] \\ \quad \langle 5, 6\rangle \\ i < m \wedge 0 \geq n \wedge \big((0 \leq n \wedge \underset{0\leq i<m}{\forall}\ \underset{0\leq l<0}{\forall} A_{\langle i,l\rangle} \neq e) \\ \qquad \vee (s = 1 \wedge A_{\langle i,0\rangle} = e)\big) \wedge s = 1 \Rightarrow \pi_1[i,j,s] \\ \quad \langle 5, 6, 7, 8, 14, 15, 16\rangle \\ i < m \wedge 0 \geq n \wedge \big((0 \leq n \wedge \underset{0\leq i<m}{\forall}\ \underset{0\leq l<0}{\forall} A_{\langle i,l\rangle} \neq e) \\ \qquad \vee (s = 1 \wedge A_{\langle i,0\rangle} = e)\big) \wedge s \neq 1 \wedge \pi_1[i+1,j,s] \\ \qquad \Rightarrow \pi_1[i,j,s] \\ \quad \langle 5, 6, 7, 8, 14, 15, 17\rangle \\ i < m \wedge j \geq n \wedge \iota_2 \wedge \big((0 \leq j \leq n \\ \qquad \wedge \underset{0\leq i<m}{\forall}\ \underset{0\leq l<j}{\forall} A_{\langle i,l\rangle} \neq e) \vee (s = 1 \wedge A_{\langle i,j\rangle} = e)\big) \\ \qquad \wedge (s = 1) \Rightarrow \pi_1[i,j,s] \\ \quad \langle 5, 6, 8, 9, 14, 15, 17\rangle \\ i < m \wedge j \geq n \wedge \iota_2 \wedge \big((0 \leq j \leq n \\ \qquad \wedge \underset{0\leq i<m}{\forall}\ \underset{0\leq l<j}{\forall} A_{\langle i,l\rangle} \neq e) \vee (s = 1 \wedge A_{\langle i,j\rangle} = e)\big) \\ \qquad \wedge s \neq 1 \wedge \pi_1[i+1,j,s] \Rightarrow \pi_1[i,j,s] \\ \quad \langle 5, 6, 8, 9, 14, 15, 16\rangle \end{cases}$$
$$\Rightarrow \underset{i,j,s:\iota_1}{\forall} \pi_2[i,j,s]$$

Termination of the inner loop.

$$\underset{j,s:\iota_2}{\forall} \wedge \begin{cases} j \geq n \Rightarrow \pi_2[j,s] \\ \quad \langle 8, 9\rangle \\ j < n \wedge (e = A_{\langle i,j\rangle}) \Rightarrow \pi_2[j,1] \\ \quad \langle 8, 9, 10, 11, 12\rangle \\ j < n \wedge (e \neq A_{\langle i,j\rangle}) \wedge \pi_2[j+1,s] \Rightarrow \pi_2[j,s] \\ \quad \langle 8, 9, 10, 13\rangle \end{cases}$$
$$\Rightarrow \underset{j,s:\iota_2}{\forall} \pi_2[j,s]$$

V. Correctness of the Method

The loop semantics as given by the function Σ can be brought into the form $\Sigma[W]$:

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} Q[\delta] \Rightarrow (f[\delta] = S[\delta]) \\ \neg Q[\delta] \Rightarrow (f[\delta] = f[R[\delta]]) \end{array} \right. \tag{3}$$

where $Q[\delta]$ represents the disjunction of the negated loop condition with all path conditions on which the loop terminates abruptly or normally, while S and R express the effects of the loop body on different paths, possibly including case distinction.

Then the termination condition given by Θ can be expressed as:

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} Q[\delta] \Rightarrow \pi[\delta] \\ \neg Q[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \tag{4}$$

The total correctness formula for `while` loops is expressed as: "The formula $\underset{\delta:\iota}{\forall} \iota[f[\delta]]$ is a logical consequence of the semantics $\Sigma[W]$ and the verification conditions." However, this always holds in the case that $\Sigma[W]$ is contradictory to

the theory, which may happen on the iterative execution path. Therefore, one proves first that the existence (and the uniqueness) of an f satisfying $\Sigma[W]$ 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 the basic theory of natural numbers (including the induction principle), and we denote by n, m natural numbers.

A first step is the general fact of the existence of the repetition function, that is the formula:

$$\underset{g}{\forall}\underset{G}{\exists}\underset{x}{\forall}(G[0,x]=x)\wedge\ (\underset{n}{\forall}G[n^+,x]=g[G[n,x]]),$$

where n^+ stands for the successor of n in the theory of natural numbers. The proof is based on natural number induction and is presented in detail in [10], and we will employ the usual notation $g^n[x]$ for $G[n,x]$, as well as the straightforward property $g^{n^+}[x]=g^n[g[x]]=g[g^n[x]]$.

Lemma 1: (Existence of the recursion index.) The formula $\underset{\delta:\iota}{\forall}\ \underset{n:\mathbb{N}}{\exists}Q[R^n[\delta]]$ is a logical consequence of the termination condition (4) and the safety verification conditions.

Proof: The proof uses the induction principle given in (4), where $\pi[\delta]$ is $\underset{n:\mathbb{N}}{\exists}Q[R^n[\delta]]$.
We have to prove that:

$$\underset{\delta:\iota}{\forall}\wedge\left\{\begin{array}{l} Q[\delta]\Rightarrow\underset{n:\mathbb{N}}{\exists}Q[R^n[\delta]] \\ \neg Q[\delta]\wedge\underset{n:\mathbb{N}}{\exists}Q[R^n[R[\delta]]]\Rightarrow\underset{n:\mathbb{N}}{\exists}Q[R^n[\delta]]\end{array}\right.$$

In the first clause n is 0, and in the second clause n on the right hand side is the successor of n from the left hand side. ∎

Remark 2: One can define now a function (the recursion index of δ) $M[\delta]=\min\{n\mid Q[R^n[\delta]\}$ because the set is nonempty.

Remark 3: It is straightforward to show that $M[R[\delta]]^+=M[\delta]$.

Theorem 1: (Existence of the function implemented by the loop.) The existence of an f satisfying formula (3) is a logical consequence of the termination condition (4) and the safety verification conditions.

Proof: We take $f[\delta]:=S[R^{M[\delta]}[\delta]]$ as the witness for the loop semantics. For showing that f satisfies (3), in the inductive clause we use the remark above. ∎

Remark 4: The uniqueness is easily proven by taking $f_1[\delta]=f_2[\delta]$ as $\pi[\delta]$ for two arbitrary functions f_1, f_2 which satisfy (3).

Theorem 2: (Total correctness.) The formula $\underset{\delta:\iota}{\forall}\iota[f[\delta]]$ is a logical consequence of the program semantics and the verification conditions.

Proof: By taking $\pi[\delta]$ as $\iota[f[\delta]]$ in (4), we have to prove that:

$$\underset{\delta:\iota}{\forall}\wedge\left\{\begin{array}{l} Q[\delta]\Rightarrow\iota[f[\delta]] \\ \neg Q[\delta]\wedge\iota[f[R[\delta]]]\Rightarrow\iota[f[\delta]]\end{array}\right.$$

The first implication holds by using the semantics definition for f in the case $Q[\delta]$ holds and the functional verification condition expressing the fact that the invariant is preserved also at the last iteration of the loop. The premises of the second implication hold because $f[R[\delta]]$ is defined ($R[\delta]$ satisfies ι) and $\iota[f[R[\delta]]]$ is preserved at each loop iteration by the corresponding functional verification condition. ∎

Note that this proof is basically identical for tail recursive functions in general, and very similar to the single recursion programs [11].

VI. Conclusion

The method presented in this paper combines forward symbolic execution and functional semantics for reasoning about [abrupt terminating] imperative non-recursive programs. A distinctive feature of our approach is the formulation of the termination condition as an induction principle depending on the structure of the program with respect to loops. Moreover, the total correctness condition is expressed at object level, and we do not need any additional construction for the definition of program execution (in particular for termination).

We are currently working at the automated proofs from Section V in the framework provided by the Theorema system [5]. As future work, we plan to extend our approach to multiple recursion programs.

In the past we approached the problem of termination of recursive programs [8], which is more complex but it is solved in a similar manner as in this paper. This approach can be easily extended to programs containing loops, but where recursive calls are outside the loops. Otherwise, mutual recursion occurs and this requires further investigation.

References

[1] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development*, Springer-Verlag, 2004.

[2] R. Boute, *Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus*, ACM Transactions on Programming Languages and Systems **28** (2006), no. 4, 747–793.

[3] 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.

[4] A. Bradley, Z. Manna, and H. Sipma, *Termination analysis of integer linear loops*, CONCUR 2005 (London, UK), Springer-Verlag, 2005.

[5] 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.

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, R. Leino, and E. Poll, *An Overview of JML Tools and Applications*, International Journal on Software Tools for Technology Transfer **7** (2005), no. 3, 212–232.

[7] B. Cook, A. Podelski, and A. Rybalchenko, *Termination Proofs for Systems Code*, ACM SIGPLAN Notices **41** (2006), no. 6, 415–426.

[8] 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.

[9] 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.

[10] 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.

[11] M. Eraşcu and T. Jebelean, *A Purely Logical Approach to Program Termination*, Proceedings of the 11th International Workshop on Termination, FLOC 2010 (P. Schneider-Kamp, ed.), July 14-15 2010.
[12] M. Gordon and T. Melham (eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, New York, NY, USA, 1993.
[13] D. Gries, *The Science of Programming*, Springer, 1981.
[14] M. Kaufmann, J. Strother Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[15] J. King, *Symbolic Execution and Program Testing*, Communications of the ACM **19** (1976), no. 7, 385–394.
[16] A. Krauss, *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*, Ph.D. thesis, Technische Universität München, 2009.
[17] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, July 2002.
[18] C. S. Lee, N. Jones, and A. Ben-Amram, *The Size-Change Principle for Program Termination*, SIGPLAN Not. **36** (2001), no. 3, 81–92.
[19] J. McCarthy, *A Basis for a Mathematical Theory of Computation*, Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.), North-Holland, Amsterdam, 1963, pp. 33–70.
[20] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, Lecture Notes in Computer Science, vol. 828, Springer, 1994.
[21] A. Podelski and A. Rybalchenko, *A Complete Method for the Synthesis of Linear Ranking Functions*, VMCAI, 2004, pp. 239–251.
[22] W. Schreiner, *Understanding Programs*, Tech. report, Research Institute for Symbolic Computation, July 2008.
[23] K. Slind, *Function Definition in Higher-Order Logic*, TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (London, UK), Springer-Verlag, 1996, pp. 381–397.
[24] S. Wolfram, *The Mathematica Book. Version 5.0*, Wolfram Media, 2003.