A Purely Logical Approach to Imperative Program Verification

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

Abstract

We present a method for the generation of the verification conditions for the total correctness of imperative programs containing nested loops with abrupt termination statements, and we illustrate it on several examples. 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 branches 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.

1 Introduction

We present the theoretical foundations of a formal method for handling the total correctness of arbitrarily-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. 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).

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 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*. (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

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

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 conditions, functional correctness conditions, and assertive conditions (conditions whose goal is represented by an intermediary program assertion), altogether ensuring the partial correctness. Separately, the *termination condition* is generated as an induction principle. Remarkably, all the conditions are expressed in the object logic.

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 items take the program text, the specification (input and output conditions) and, eventually, assertions like the loop invariant or others, introduced by `Assert` construct. The program can be annotated via `Assert` at any point of the program and mandatory after a loop terminating abruptly via `break`. We impose the second case because, at the end of the loop terminated via `break`, 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.

The approach is also universal because the programming language itself is predicate logic (terms and formulae from the object theory), except for `Assert` construct, and the few meta-constructs which represent the basic imperative statements (assignment, conditionals, loops, abrupt statements: `break`, `return`).

The method is implemented in Mathematica [22] 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 meta-level definitions presented in the following sections and the real implementation are minor.

This paper extends the calculus presented in [8] and details the approach for programs containing `while` loops. Emphasis is put on the termination of loops. We also 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 and we think that this set is minimal. Proofs in the *Theorema* system [5] will be done in order to confirm or to extend this assumption as part of the future work.

Related Work

Our approach follows the principles of forward symbolic execution [13] and functional semantics [17], 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; [15] 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, [20] presents the formal calculus for imperative languages containing complex structures. Specification languages used in the framework of verification tools use also 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 [11], to synthesize the termination term based on the loop behavior [4, 19, 3] 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` constructs). But actually we transform the imperative program into a functional one (a tail-recursive function for each loop), so one expects to apply some kind of operational (call-by-value or call-by-name) or denotational semantics (based on fix-point theory) for functional languages. One 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, no matter if the program is imperative or recursive.*

Most of the proof assistants provide infrastructure for proving/disproving the termination of classical examples with general recursion. ACL2 [12] handles total functions that must be proved total at the definition time; sometimes the system is able to infer this fact. Isabelle [18], HOL4 [10] and Coq [1] are basically using the recursion package TFL [21] 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 [14] 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 [16] or the approaches for proving the termination of industrial-size code (Microsoft Windows Operating System Drivers) [7].

2 Logical Foundations of Imperative Loops

Our approach is purely logic, meaning that the program correctness, and implicitly loop correctness, are 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. 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, $\smile$ is the concatenation of tuples; $x \rightarrow t$ 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: $v \in V \subset \mathscr{V}$ is an initialized variable, $t \in \mathscr{T}$ is a term, φ, ϑ are boolean expressions, 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 and ι' denote conventionally loop invariants which hold at the beginning of the loop and are inductively preserved by each iteration of the loop. The user should provide such an invariant. We denote conventionally by δ's the critical variables. For simplicity we consider a single critical variable δ in loops. 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- (I_P)and postconditions (O_P). It takes as input a certain number of variables, conventionally denoted α and it returns a single value, conventionally denoted β.

The meta-theory also contains the properties of the meta-predicates Π, Π' (syntax checkers) and meta-functions Σ, Σ', and Σ'' (semantics constructors), Γ and Γ' (verification conditions generators), 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 conditions 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.
- an assertion introduced via `Assert` should follow from the current path condition.

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, considering that 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, or, in the case of loops terminated abruptly via `break`, they are assumed to fulfill the invariant and the assertion after the loop.

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

Note that 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.

Algorithm 1 Search in a bidimensional array

```
1. $\underline{\texttt{in}}\, A$:$\underline{\texttt{array of reals}}$, $m,n$:$\underline{\texttt{integers}}$, $e$:$\underline{\texttt{real}}$ where $m \geq 0, n \geq 0$
2. $\underline{\texttt{out}}\, \beta$: $\underline{\texttt{integer or tuple where}}$ $(\underset{0\leq k<m}{\exists}\underset{0\leq l<n}{\exists} A_{[k][l]} = e \Rightarrow (A_{[\beta_1][\beta_2]} = e))\wedge$
   $(\underset{0\leq k<m}{\forall}\underset{0\leq l<n}{\forall} A_{[k][l]} \neq e \Rightarrow (\beta = -1))$
3. $\underline{\texttt{local int}}\; i,j$;
4. $i := 0$;
5. $\underline{\texttt{while}}\; (i < m)$
6.     $\underline{\iota_1}:\; 0 \leq i \leq m \wedge 0 \leq j \leq n \wedge \underset{0\leq k<i}{\forall}\underset{0\leq j<n}{\forall} A_{[k][j]} \neq e,$
7.     $j := 0$;
8.     $\underline{\texttt{while}}\; (j < n)$
9.       $\underline{\iota_2}: 0 \leq i \leq m \wedge 0 \leq j \leq n \wedge \underset{0\leq k<i}{\forall}\underset{0\leq l<j}{\forall} A_{[k][l]} \neq e,$
10.      $\underline{\texttt{if}}\; (e = A_{[i][j]})$
11.         $\underline{\texttt{return}}[\langle i,j\rangle]$;
12.      $j := j+1$;
13.    $i := i+1$;
14.$\underline{\texttt{return}}$[-1]
```

3 Syntax and Semantics

We first introduce a meta-predicate Π (and the auxiliary one Π') which checks the syntax and a meta-function Σ (and the auxiliary ones Σ' and 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 construct 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 (and the auxiliary ones Γ'), while the termination condition, one for each `while` loop is generated by the meta-level function Θ (and the auxiliary ones Θ', Θ'').

3.1 Syntax

The predicate Π checks a program for syntactic correctness including the fact that each variable is initialized, that each branch has a `return` statement and that `break` statement occurs only in `while` loops and, if that is the case, then an `Assert` construct occurs after the loop body. The meta-level function *Vars* constructs a list with the variables occurring in a term or formula, the meta-level predicate *IsFOLFormula* checks whether an expression is a first-order logic formula and *HasAssert* checks if, in case of `break` statement, the program is annotated with an `Assert` construct.

Definition 1. *1.* $\Pi[P] :\Leftrightarrow \bigwedge \begin{cases} IsFOLFormula[I_P[\alpha]] \\ IsFOLFormula[O_P[\alpha,\beta]] \\ \Pi[\{\alpha \to \alpha_0\}, P]\{\alpha_0 \to \alpha\} \end{cases}$

2. $\Pi[V, \langle \underline{\texttt{return}}[t]\rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V$

3. $\Pi[V, \langle v:\; = t\rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V \wedge \Pi[V \cup \{v\}, P]$

4. $\Pi[V, \langle \underline{\texttt{if}}(\varphi)P_T, P_F\rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \\ \Pi[V, P_T \smile P] \\ \Pi[V, P_F \smile P] \end{cases}$

5. $\Pi[V, \langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \wedge IsFOLFormula[\iota] \\ \Pi'[V, B \smile P] \\ \Pi[V, P] \end{cases}$

6. $\Pi[V, \langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P] :\Leftrightarrow IsFOLFormula[\varphi] \wedge \Pi[V, P]$

7. $\Pi[V, P] = \mathbb{F}$

The input variable α from Definition 1.1 behaves like a global variable. Some of the principles of the syntactic check are as follows: the variables occurring in a term t or in a formula φ have to be initialized (e.g. Definition 1.2). The formulae I_P, O_P, φ, and ι, which are supposed to be well-formed first-order formulae, are checked correspondingly (e.g. Definition 1.5). In case of successful assignment the variable v is added to the set V of initialized variables. `break` statement outside the loop body gives a syntax error (Definition 1.7), while it is allowed inside the loop body (Definition 2). This is also the reason why a new auxiliary predicate Π' was introduced to make distinction between its behavior. Moreover if `break` occurs, we also check if the program is annotated with an `Assert` construct (Definition 2.7). Absence of the `return` statement in the program means syntax error (Definition 1.7), unlike the same situation in the loop body (Definition 2.5), which means that we reached the end of the loop body.

Definition 2. *1.* $\Pi'[V, \langle \underline{\texttt{return}}[t]\rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V$

2. $\Pi'[V, \langle \underline{\texttt{break}}\rangle \smile P] :\Leftrightarrow HasAssert[P]$

3. $\Pi'[V, \langle v :\, = t\rangle \smile P] :\Leftrightarrow Vars[t] \subseteq V \wedge \Pi'[V \cup \{v\}, P]$

4. $\Pi'[V, \langle \underline{\texttt{if}}(\varphi)P_T, P_F\rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \\ \Pi'[V, P_T \smile P] \\ \Pi'[V, P_F \smile P] \end{cases}$

5. $\Pi'[V, \langle\rangle] :\Leftrightarrow \mathbb{T}$

6. $\Pi'[V, \langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P] :\Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ IsFOLFormula[\varphi] \wedge IsFOLFormula[\iota] \\ \Pi'[V, B \smile P] \\ \Pi'[V, P] \end{cases}$

7. $\Pi'[V, \langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P] :\Leftrightarrow IsFOLFormula[\varphi] \wedge \Pi'[V, P]$

8. $\Pi'[V, P] :\Leftrightarrow \mathbb{F}$

The meta-predicate Π', except for `break` statement, behaves similarly to Π.

Remark 1. *We are aware that the syntax checker could be implemented by a single meta-predicate and additionally using a global variable to check whether we are or we are not in a loop with* `break` *statement. We prefer however specialized auxiliary predicate for the loop body for the cleanliness of the formalization. The preference for specialized functions for [different types of] loops will be encountered also in the formalization of the meta-level functions in the next sections.*

For instance our illustrating example is syntactically correct.

3.2 Semantics

For programs containing `while` loops there are two possibilities for constructing their semantics. The one presented in [8] 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 does not allow to reason about the loop itself, because its effect is encoded into the invariant, rather than in its semantics. We propose in this paper a semantics relating `while` loops to function calls (2), which is more precise and necessary for proving the correctness of our method in Section 6.

We define the semantics of a program via the meta-level function Σ as being an implicit definition *at object level* of the function implemented by the program. The semantics function depends also on some global variables like the initial program state, the input condition, and symbols $\mathscr{F}$ – standing for the program function, respectively f_i ($i \in \mathbb{N}$) – a new symbol for each loop analyzed. These denote conventionally the function defined by the loop and, together with a tuple of statements, are the arguments of the meta-level semantics function. The output is a list of the formulae with the shape:

$$\underset{\alpha:I_P}{\forall}\big(\Phi \Rightarrow (\mathscr{F}[\alpha] = t)\big), \tag{1}$$

one for each path of the program. In case of loop semantics, $\mathscr{F}$ is actually a f. Each formula is a conditional definition for $\mathscr{F}[\alpha]$ and depends on the accumulated [negated] conditions Φ coming from the `if` statements and from the effect of the loop encoded as the invariant leading to a certain `return` statement, whose argument (symbolically evaluated) represents the corresponding value of $\mathscr{F}[\alpha]$, namely t. The operations from the body of the loops present in the program are not reflected explicitly in these clauses, except if a `return` or a `break` in non-nested loops is encountered. Similarly, the operations performed in a nested loop are not visible in its wrapper loop (Definition 5.5), except the outcome of the abrupt statements. It is the loop invariant that encodes them. Each loop occurring at the main level will trigger the generation of two paths – leading to two sets of clauses. On one path the loop is not executed, and on the other the loop is executed and its effect is expressed using the invariant. The loops occurring in nested way are not visible at all in the semantics, but their effect will be used in the generation of the verification conditions.

For example the branch $\langle 1,3,5,6,8,9,10,11\rangle$ of the *Example 1* considers the execution of both loops, which are terminating abruptly. On this branch the semantics of the program is: $m \geq 0 \wedge n \geq 0 \wedge i < m \wedge \iota_1 \wedge j < n \wedge \iota_2 \wedge (e = A_{[i][j]}) \Rightarrow (\mathscr{F}[m,n] = \langle i,j\rangle)$. Note that the `return` statement influences also the semantics of the loops, namely, on the branch $\langle 5,6,8,9,10,11\rangle$ the outer loop has the semantics: $\iota_1 \wedge i < m \wedge j < n \wedge \iota_2 \wedge (e = A_{[i][j]}) \Rightarrow (f_1[i,j] = \langle i,j\rangle)$, and on the path: $\langle 8,9,10,11\rangle$, the inner loop has the semantics $\iota_2 \wedge j < n \wedge (e = A_{[i][j]}) \Rightarrow (f_2[j] = \langle i,j\rangle)$.

Template semantics formula for loops (2) states the following: for all critical variables satisfying the invariant, if the loop condition is not fulfilled then its semantics (denoted conventionally by f) can be computed directly in terms of the function S from the object theory; otherwise the loop analysis leads to recursive definitions of the semantics on each branch, the recursive argument being constructed by the application of the substitution σ_W to the critical variables. The substitution σ_W is obtained by updating σ_0 with all the assignments on the respective path of the loop.

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} \neg\varphi \Rightarrow (f[\delta] = S[\delta]) \\ \varphi \wedge \Phi \Rightarrow (f[\delta] = f[\delta\sigma_W]) \\ \ldots \end{cases} \tag{2}$$

The meta-level function Σ together with the auxiliary functions Σ' and Σ'' produce implicit definitions for each loop of the program and also for the program itself.

Definition 3. *1.* $\Sigma[P] = \Sigma[\{\alpha \to \alpha_0\}, I_P[\alpha_0], P, \mathscr{F}]\{\alpha_0 \to \alpha\}$

2. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P, \mathscr{F}] = \langle \Phi \Rightarrow (\mathscr{F}[\alpha_0] = t\sigma)\rangle$

3. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, \mathscr{F}] = \langle\rangle$

4. $\Sigma[\sigma, \Phi, \langle v := t\rangle \smile P, \mathscr{F}] = \Sigma[\sigma\{v \to t\sigma\}, \Phi, P, \mathscr{F}]$

5. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{if}}(\varphi)P_T, P_F\rangle \smile P, \mathscr{F}] = \smile \left\{ \begin{array}{l} \Sigma[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \mathscr{F}] \\ \Sigma[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \mathscr{F}] \end{array} \right.$

6. $\Sigma[\sigma, \Phi, \langle\rangle, \mathscr{F}] = \langle\rangle$

7. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P, \mathscr{F}] =$

$$\smile \left\{ \begin{array}{ll} \Sigma[\sigma, \Phi \wedge \neg\varphi\sigma, P, \mathscr{F}] & (1) \\ \left\langle \underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} (\neg\varphi\sigma_0 \Rightarrow (f[\delta] = \delta\sigma))\{\delta_0 \to \delta\} \\ \Sigma'[\sigma_0, \varphi\sigma_0, B, f]\{\delta_0 \to \delta\} \end{array} \right\} \right\rangle & (2) \\ \Sigma[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B, \mathscr{F}] & (3) \\ \Sigma[\sigma_0, \Phi \wedge \neg\varphi\sigma_0 \wedge \iota\sigma_0, P, \mathscr{F}] & (4) \end{array} \right.$$

8. $\Sigma[\sigma, \Phi, \langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P, \mathscr{F}] = \Sigma[\sigma, \Phi \wedge \varphi\sigma, P, \mathscr{F}]$

The meta-level function Σ constructs a set of implicit definitions for the current module. The basic idea is to start with the a substitution which assigns the conventional constant[s] α_0 to the program argument[s] α and with the input condition as a premise, and then to update the current substitution and the current path condition according to the statements of the program (symbolic execution).

A `return` statement constructs the expression of the program function on the respective branch (Definition 3.2), assignments update the current state of the program (Definition 3.4), `if` forks the program execution (Definition 3.5). Definitions 3.3 (`break`) and 3.6 (end of the loop) are applied only in the case the currently analyzed module has no nested loops. Analysis of a `while` loop is done as follows: one branch considers that the loop is not executed at all (Definition 3.7.1), respectively, that the loop was analyzed and terminates (Definition 3.7.4) and in both cases proceeds with the execution of the rest of the program. One branch analyzes the loop body after the same principles the program is (Definition 3.7.3). Additionally semantics formula for the current loop is constructed (Definition 3.7.2), where the auxiliary function Σ' is used (Definition 4). `break` behavior is different for non-nested and nested loops: for non-nested loops in the current module analyzed it computes the program function on the respective branch (Definition 4.2), in contrary, has no effect for nested loops (Definition 5.2). Definition 3.7.3 analyzes inductively the structure of `while` loops handling in this way arbitrarily nested loops. Each time Definition 3.7.2 is applied a new symbol f denoting conventionally the function of the current loop is generated which is used also in and Definition 3.7.3. An `Assert` construct (Definition 3.8) updates the set of assumptions and afterwards continues the analysis of the program.

The inductive definitions of the auxiliary semantics functions are similar to the main function, except for `break` and end of the loop body when [recursive] semantics definitions have to be constructed.

Definition 4. *1.* $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{return}}[t]\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = t\sigma)\big)$

2. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{break}}\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = \delta_0\sigma)\big)$

3. $\Sigma'[\sigma, \Phi, \langle v := t\rangle \smile P, f] = \Sigma'[\sigma\{v \to t\sigma\}, \Phi, P, f]$

4. $\Sigma'[\sigma, \Phi, \langle \underline{\texttt{if}}(\varphi)P_T, P_F\rangle \smile P, f] = \wedge \left\{ \begin{array}{l} \Sigma'[\sigma, \Phi, P_T \smile P, f] \\ \Sigma'[\sigma, \Phi, P_F \smile P, f] \end{array} \right.$

5. $\Sigma'[\sigma,\Phi,\langle\rangle,f] = \big(\Phi \Rightarrow (f[\delta_0] = f[\delta\sigma]\big)$

6. $\Sigma'[\sigma,\Phi,\langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P, f] = \Sigma''[\sigma,\Phi,\langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P, f]\{\delta_0 \to \delta\}$

7. $\Sigma'[\sigma,\Phi,\langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P, f] = \Sigma[\sigma,\Phi\wedge\varphi\sigma,P,f]$

Definition 5. *1.* $\Sigma''[\sigma,\Phi,\langle \underline{\texttt{return}}[t]\rangle \smile P, f] = \big(\Phi \Rightarrow (f[\delta_0] = t\sigma)\big)$

2. $\Sigma''[\sigma,\Phi,\langle \underline{\texttt{break}}\rangle \smile P, f] = \mathbb{T}$

3. $\Sigma''[\sigma,\Phi,\langle v := t\rangle \smile P, f] = \Sigma''[\sigma\{v \to t\sigma\},\Phi,P,f]$

4. $\Sigma''[\sigma,\Phi,\langle \underline{\texttt{if}}(\varphi)P_T,P_F\rangle \smile P, f] = \wedge\begin{cases} \Sigma''[\sigma,\Phi\wedge\varphi\sigma,P_T \smile P,f] \\ \Sigma''[\sigma,\Phi\wedge\neg\varphi\sigma,P_F \smile P,f] \end{cases}$

5. $\Sigma''[\sigma,\Phi,\langle\rangle,f] = \mathbb{T}$

6. $\Sigma''[\sigma,\Phi,\langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota, B\rangle \smile P, f] = \wedge\begin{cases} \Sigma'[\sigma,\Phi\wedge\neg\varphi\sigma,P,f] \\ \Sigma''[\sigma_0,\varphi\sigma_0\wedge\iota\sigma_0,B,f] \\ \Sigma'[\sigma_0,\Phi\wedge\neg\varphi\sigma_0\wedge\iota\sigma_0,P,f] \end{cases}$

7. $\Sigma''[\sigma,\Phi,\langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P, f] = \Sigma''[\sigma,\Phi\wedge\varphi\sigma,P,f]$

Remark 2. *We are aware that the semantics constructor could be implemented with less meta-level functions and having some supplementary global variables. We prefer however specialized auxiliary predicate for the loop body for the cleanliness of the formalization.*

The semantics formulae of *Example 1* are as follows.

$\underline{\text{Semantics of the program.}}$

$m \geq 0 \wedge n \geq 0 \wedge 0 \geq m \Rightarrow (\mathscr{F}[m,n] = -1)$

$i < m \wedge \iota_1 \wedge j < n \wedge \iota_2 \wedge (e = A_{[i][j]}) \Rightarrow (\mathscr{F}[m,n] = \langle i,j\rangle)$

$m \geq 0 \wedge n \geq 0 \wedge i \geq m \wedge \iota_1 \Rightarrow (\mathscr{F}[m,n] = -1)$

$\underline{\text{Semantics of the outer loop.}}$

$$\underset{i,j:\iota_1}{\forall} \wedge \begin{cases} i \geq m \Rightarrow (f_1[i,j] = \langle i,j\rangle) \\ i < m \wedge j \geq n \Rightarrow (f_1[i,j] = f_1[i+1,j]) \\ i < m \wedge j < n \wedge \iota_2 \wedge (e = A_{[i][j]}) \Rightarrow (f_1[i,j] = \langle i,j\rangle) \\ i < m \wedge j \geq n \wedge \iota_2 \Rightarrow (f_1[i,j] = f_1[i+1,j]) \end{cases}$$

$\underline{\text{Semantics of the inner loop.}}$

$$\underset{j:\iota_2}{\forall} \begin{cases} j \geq n \Rightarrow (f_2[j] = j) \\ j < n \wedge (e = A_{[i][j]}) \Rightarrow (f_2[j] = \langle i,j\rangle) \\ j < n \wedge (e \neq A_{[i][j]}) \Rightarrow (f_2[j] = f_2[j+1]) \end{cases}$$

4 Generation of the Partial Correctness Verification Conditions

The meta-functions Γ and Γ' generate 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). Separately, the meta-function Θ generates a termination condition for each loop. (We do not treat in this paper recursive calls, thus no termination condition is generated for the main program.)

The meta-level function Γ generates safety and functional verification conditions. *Safety conditions* are formulae with the shape $\Phi \Rightarrow I_h[t]$, where I_h is the input condition of some function h, and t the

symbolic value of the function call (the formula Φ accumulates the conditions from `if` and `while` statements and the input and output conditions for the function calls on the respective branch). For instance the precondition (*True*) of $+$ from the assignments of our running example has to be ensured. More interesting examples are on bounded integers, where the safety conditions can detect arithmetic overflow. *Functional and, respectively, assertive conditions* are formulae checking that the output condition on the currently returned value, respectively, the assertion at a certain point in the program, is a consequence of the accumulated conditions on the respective branch. An example of functional verification condition is the formula:
$m \geq 0 \wedge n \geq 0 \wedge 0 \geq m \Rightarrow (\underset{0 \leq k<m}{\exists} \underset{0 \leq l<n}{\exists} A_{[k][l]} = e \Rightarrow (A_{[\beta_1][\beta_2]} = e)) \wedge (\underset{0 \leq k<m}{\forall} \underset{0 \leq l<n}{\forall} A_{[k][l]} \neq e \Rightarrow (-1 = -1))$, obtained from the analysis of the path: $\langle 1,3,4,5,14,2 \rangle$.

As in the previous definitions, the function Γ depends also on some global variables, namely, the initial program state and the input specification. The arguments are the current symbolic state σ, the accumulated path conditions Φ and a tuple of statements, and generates a list of verification conditions.

Definition 6. *1.* $\Gamma[P] = \Gamma[\{\alpha \to \alpha_0\}, I_P[\alpha_0], P]\{\alpha_0 \to \alpha\}$

2. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{return}}[t] \rangle \smile P] = \langle \Phi \Rightarrow O_f[\alpha_0, t\sigma] \rangle$

3. $\Gamma[\sigma, \Phi, \langle v := t \rangle \smile P] = \Gamma[\sigma\{v \to t\sigma\}, \Phi, P]$

4. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{if}}(\varphi) P_T, P_F \rangle \smile P] = \smile \begin{cases} \Gamma[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P] \\ \Gamma[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P] \end{cases}$

5. $\Gamma[\sigma, \Phi, \langle \rangle] = \langle \Phi \Rightarrow \iota\sigma \rangle$

6. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{while}}(\varphi) \underline{\texttt{do}} \iota, B \rangle \smile P] = \smile \begin{cases} \Gamma[\sigma, \Phi \wedge \neg\varphi\sigma, P] & (1) \\ \langle \Phi \Rightarrow \iota\sigma \rangle & (2) \\ \Gamma'[\sigma_0, \iota\sigma_0 \wedge \varphi\sigma_0, \iota, B]\{\delta_0 \to \delta\} & (3) \\ \Gamma[\sigma_0, \iota\sigma_0 \wedge \neg\varphi\sigma_0, P] & (4) \end{cases}$

7. $\Gamma[\sigma, \Phi, \langle \underline{\texttt{Assert}}[\varphi] \rangle \smile P] = \smile \begin{cases} \langle \Phi \Rightarrow \varphi\sigma \rangle \\ \Gamma[\sigma, \Phi \wedge \varphi\sigma, P] \end{cases}$

The input variable α behaves as a global variable. The verification conditions for partial correctness are generated as follows: the `return` statement finishes the analysis of the respective branch, checking that the postcondition (Definition 6.2) is fulfilled, an assignment updates the the program state (Definition 6.3), `if` forks the program analysis (Definition 4). `while` loops split the analysis of the program in three branches: one branch considers that the loop is not executed (Definition 6.6.1), another analyzes the body of the loop (Definition 6.6.3) after checking that the invariant holds at the beginning of the loop analysis (Definition 6.6.2), the third one analyzes the rest of the program taking into account that the loop was executed (the invariant and the negated loop condition are part of the path condition) and terminates (Definition 6.6.4). The assertion introduced via `Assert` is checked if it follows from the existing assumptions 6.7 and afterwards it is added to the accumulated assumptions on the respective path. Note that due to the syntax check which does not allow a `break` in the main program, no corresponding inductive definition is needed.

Definition 7. *1.* $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{return}}[t] \rangle \smile P] = \langle \Phi \Rightarrow O_f[\alpha_0, t\sigma] \rangle$

2. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle \underline{\texttt{break}} \rangle \smile P] = \langle \Phi \Rightarrow \vartheta\sigma \rangle$

3. $\Gamma'[\sigma, \Phi, \iota, \varphi, \langle v := t \rangle \smile P] = \Gamma'[\sigma\{v \to t\sigma\}, \Phi, \iota, \varphi, P]$

4. $\Gamma'[\sigma,\Phi,\iota,\varphi,\langle \underline{\texttt{if}}(\varphi)P_T,P_F\rangle \smile P] = \smile \left\{ \begin{array}{l} \Gamma'[\sigma,\Phi\wedge\varphi\sigma,\iota,\varphi,P_T \smile P] \\ \Gamma'[\sigma,\Phi\wedge\neg\varphi\sigma,\iota,\varphi,P_F \smile P] \end{array}\right.$

5. $\Gamma'[\sigma,\Phi,\iota,\varphi,\langle\rangle] = \langle \Phi \Rightarrow \iota\sigma \rangle$

6. $\Gamma'[\sigma,\Phi,\iota,\varphi,\langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota',B\rangle \smile P] = \smile \left\{ \begin{array}{l} \Gamma'[\sigma,\Phi\wedge\neg\varphi\sigma,\iota,\varphi,P] \\ \langle \Phi \Rightarrow \iota\sigma \rangle \\ \Gamma'[\sigma_0,\iota'\sigma_0\wedge\varphi\sigma_0,\iota_1,\varphi,B]\{\delta_0 \to \delta\} \\ \Gamma'[\sigma_0,\iota_1\sigma_0\wedge\neg\varphi\sigma_0,\iota_1,\varphi,P] \end{array}\right.$

7. $\Gamma'[\sigma,\Phi,\iota,\varphi,\langle \underline{\texttt{Assert}}[\vartheta]\rangle \smile P] = \smile \left\{ \begin{array}{l} \langle \Phi \Rightarrow \vartheta\sigma \rangle \\ \Gamma'[\sigma,\Phi\wedge\vartheta\sigma,\iota,\varphi,P] \end{array}\right.$

The inductive definitions corresponding to Γ' are the same as of Γ, except that `break` also occurs. The symbol ϑ stands for an assertion and it is a first order logic formula. The `Assert` construct occurs in the tuple of statements P, so Definition 7.2 is correct because ϑ occurs also in the left hand side.

Remark 3. *We are aware to the fact that only a meta-function could have been used for the generation of the verification conditions by using global variables for the extra arguments ι and φ of Γ'. Though we prefer two meta-level functions for cleanliness.*

5 Termination

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

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} \neg\varphi \Rightarrow \pi[\delta] \\ \varphi\wedge\Phi\wedge\pi[\delta\sigma_W] \Rightarrow \pi[\delta] \\ \ldots \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \tag{3}$$

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. The formula consists in 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 explain now the rationale behind (3). 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 (2), the ellipsis in (3) 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 (program or loop) and specializes itself into Θ' for the analysis of loops and into Θ'' for modules which contain nested loops, because the `break` statement has different behavior for nested, respectively non-nested loops.

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 8.1) and also each time a new module is analyzed (Definition 8.7.2). 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 8.4), forks the analysis of the program in two execution paths in case of conditional `if` (Definition 8.5), 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 8.3 – handling `break` statements and 8.6 – handling end of `while` are applied only if the module analyzed does not contain nested loops, these cases occurring by the application of Definition 8.7.2. 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 8.7.1) where an auxiliary function Θ' is used, a path analyzes the loop body similarly to the program (Definition 8.7.2) and the last one continues with the analysis of the statements after the loop (Definition 8.7.3). Note that same analysis is performed to each loop, independently of the degree of nestedness, due to Definition 8.7.2. An `Assert` construct (Definition 8.8) updates the set of assumptions and afterwards continues the analysis of the program.

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

Definition 8.

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. $\Theta[\sigma, \Phi, \langle \underline{\texttt{if}}(\varphi)P_T, P_F\rangle \smile P] = \smile \begin{cases} \Theta[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P] \\ \Theta[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P] \end{cases}$

6. $\Theta[\sigma, \Phi, \langle\rangle] = \langle\rangle$

7. $\Theta[\sigma,\Phi,\langle \underline{\mathtt{while}}(\varphi)\underline{\mathtt{do}}\iota, B \smile P] =$
$$\smile \left\{ \begin{array}{l} \left\langle \underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} (\neg\varphi\sigma_0 \Rightarrow \pi[\delta])\{\delta_0 \to \delta\} \\ \Theta'[\sigma_0, \varphi\sigma_0, B, \pi]\{\delta_0 \to \delta\} \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \right\rangle \quad (1) \\ \Theta[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B] \quad (2) \\ \Theta[\sigma_0, \mathbb{T}, P] \quad (3) \end{array} \right.$$

8. $\Theta[\sigma,\Phi,\langle \underline{\mathtt{Assert}}[\varphi]\rangle \smile P] = \Theta[\sigma, \Phi \wedge \varphi\sigma, P]$

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 8.7.1) and that:

- `return` statement has the same behavior in non- and nested loops: they return the accumulated path conditions (Definitions 9.1 and 10.1);
- `break` statement behaves similarly to `return` in non-nested loops (Definition 9.2), but for programs with nested loops the analysis performed in inner loops is not visible in the wrapper ones (Definitions 10.2).
- at the end of the non-nested loop, a path condition involving the termination predicate π is constructed (Definition 9.5), while the analysis performed in the nested loops is not visible in the outer loops (Definition 10.5)
- nested loops are always analyzed by the meta-function Θ'' (Definition 9.6).

Definition 9.

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

Definition 10.

1. $\Theta''[\sigma,\Phi,\langle \underline{\mathtt{return}}[\delta]\rangle \smile P, \pi] = \Phi$
2. $\Theta''[\sigma,\Phi,\langle \underline{\mathtt{break}}\rangle \smile P, \pi] = \mathbb{F}$
3. $\Theta''[\sigma,\Phi,\langle v := t\rangle \smile P, \pi] = \Theta''[\sigma\{v \to t\sigma\}, \Phi, P, \pi]$
4. $\Theta''[\sigma,\Phi,\langle \underline{\mathtt{if}}(\varphi)P_T, P_F\rangle \smile P, \pi] = \vee \left\{ \begin{array}{l} \Theta''[\sigma, \Phi \wedge \varphi\sigma, P_T \smile P, \pi] \\ \Theta''[\sigma, \Phi \wedge \neg\varphi\sigma, P_F \smile P, \pi] \end{array} \right.$
5. $\Theta''[\sigma,\Phi,\langle\rangle, \pi] = \mathbb{F}$

6. $\Theta''[\sigma,\Phi,\langle \underline{\texttt{while}}(\varphi)\underline{\texttt{do}}\iota,B\rangle \smile P,\pi] = \vee \begin{cases} \Theta'[\sigma,\Phi\wedge\neg\varphi\sigma,P,\pi] \\ \Theta''[\sigma_0,\varphi\sigma_0\wedge\iota\sigma_0,B,\pi] \\ \Theta'[\sigma_0,\neg\varphi\sigma_0\wedge\iota\sigma_0,P,\pi] \end{cases}$

7. $\Theta''[\sigma,\Phi,\langle \underline{\texttt{Assert}}[\varphi]\rangle \smile P,\pi] = \Theta''[\sigma,\Phi\wedge\varphi\sigma,P,\pi]$

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. The line corresponding to the invariant of the currently analyzed loop occurs in all tuples but is omitted.

Termination of the outer loop.

$$\underset{i,j:\iota_1}{\forall} \wedge \begin{cases} i \geq m \Rightarrow \pi_1[i,j] \\ \langle 5,6\rangle \\ i<m\wedge j\geq n\wedge \pi_1[i+1,j] \Rightarrow \pi_1[i,j] \\ \langle 5,6,8,13\rangle \\ i<m\wedge j<n\wedge \iota_2\wedge (A_{[i][j]}=e) \Rightarrow \pi_1[i,j] \\ \langle 5,6,8,9,10\rangle \\ i<m\wedge j\geq n\wedge \iota_2\wedge \pi_1[i+1,j] \Rightarrow \pi_1[i,j] \\ \langle 5,6,8,9,13\rangle \end{cases}$$
$$\Rightarrow \underset{i,j:\iota_1}{\forall} \pi_1[i,j]$$

Termination of the inner loop.

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

Remark 4. *We are aware that the termination condition generator could be implemented with less meta-level functions and having some supplementary global variables. We prefer however specialized auxiliary predicate for the loop body for the cleanliness of the formalization.*

6 Correctness of the Method

The loop semantics as given by the function Σ can be brought into the form:

$$\Sigma[W] :\Leftrightarrow \underset{\delta:\iota}{\forall} \wedge \begin{cases} Q[\delta] \Rightarrow (f[\delta]=S[\delta]) \\ \neg Q[\delta] \Rightarrow (f[\delta]=f[R[\delta]]) \end{cases} \tag{4}$$

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:

$$\underset{\delta:\iota}{\forall} \wedge \left\{ \begin{array}{l} Q[\delta] \Rightarrow \pi[\delta] \\ \neg Q[\delta]\wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota}{\forall} \pi[\delta] \tag{5}$$

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 n,m as natural numbers.

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* (5) *and the safety verification conditions.*

Proof. The proof uses the induction principle given in (5), where $\pi[\delta]$ is $\underset{n:\mathbb{N}}{\exists} Q[R^n[\delta]]$.
We have to prove that:

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} Q[\delta] \Rightarrow \underset{n:\mathbb{N}}{\exists} Q[R^n[\delta]] \\ \neg Q[\delta] \wedge \underset{n:\mathbb{N}}{\exists} Q[R^n[R[\delta]]] \Rightarrow \underset{n:\mathbb{N}}{\exists} Q[R^n[\delta]] \end{cases}$$

In the first clause n is 0, and in the second clause n on the right hand side is the successor (in natural number theory) of n from the left hand side. Therefore $\underset{n:\mathbb{N}}{\exists} Q[R^n[\delta]]$ holds. □

Remark 5. *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 6. *It is straightforward to show that $M[\delta]^+ = M[\delta]$.*

Theorem 1. *(Existence of the function implemented by the loop.) The formula* (4) *is a logical consequence of the termination condition* (5) *and the safety verification conditions.*

Proof. We take $f[\delta] := R^{M[\delta]}[\delta]$ the witness for the loop semantics. Clearly f satisfies (4). Further we prove that:

$$\underset{m:\mathbb{N}}{\forall}\ \underset{f}{\exists}\ \underset{\delta:\iota}{\forall} (M[\delta] \leq m) \Rightarrow R^{M[\delta]}[\delta]$$

by natural induction on m.
Base Case. If $m = 0$ then $n = 0$ and $f[\delta] := R^0[\delta] = S[\delta]$.
Induction Step. Assume $\underset{f}{\exists}\ \underset{\delta:\iota}{\forall} (M[\delta] \leq m) \Rightarrow R^{M[\delta]}[\delta]$ and prove $\underset{f}{\exists}\ \underset{\delta:\iota}{\forall} (M[\delta] \leq m+1) \Rightarrow R^{M[\delta]}[\delta]$. The goal from the induction hypothesis and from the induction conclusion are identical for $M[\delta] \leq m$. In the case $M[\delta] = m+1$, the goal of the induction conclusion can be rewritten as $R^{M[\delta]^+}[\delta]$ and further as $R^{M[\delta]}[\delta]$, due to *Remark 6*. □

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 (5), we have to prove that:

$$\underset{\delta:\iota}{\forall} \wedge \begin{cases} Q[\delta] \Rightarrow \iota[f[\delta]] \\ \neg Q[\delta] \wedge \iota[f[R[\delta]]] \Rightarrow \iota[f[\delta]] \end{cases}$$

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 [9].

7 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).

In the past we approached the problem of termination of recursive programs, 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, the termination problem appears to be very complex, thus further investigation of it is needed.

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, *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.

[9] 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.), to appear, July 14-15 2010.

[10] 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.

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

[12] M. Kaufmann, J. Strother Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

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

[14] A. Krauss, *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*, Ph.D. thesis, Technische Universität München, 2009.

[15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, July 2002.

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

[17] 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.

[18] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, Lecture Notes in Computer Science, vol. 828, Springer, 1994.

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

[20] W. Schreiner, *Understanding Programs*, Tech. report, Research Institute for Symbolic Computation, July 2008.

[21] 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.

[22] S. Wolfram, *The Mathematica Book. Version 5.0*, Wolfram Media, 2003.