A Calculus for Imperative Programs: Formalization and Implementation

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

***Abstract*—As an extension of our previous work on imperative program verification, we present a formalism for handling the total correctness of `While` loops in imperative programs, consisting in functional based definitions of the verification conditions for both partial correctness and for termination. A specific feature of our approach is the generation of verification conditions as first order formulae, including the termination condition which is expressed as an induction principle.**

***Keywords*-program analysis and verification, symbolic execution, theorem proving**

I. Introduction

We present a formalism for handling the total correctness of `While` loops in imperative programs, consisting in functional based definitions of the verification conditions for both partial correctness and for termination. These definitions constitute two meta-level functions which take the program text and the loop invariant.

This is an extension of our previous work [7], in which we present a method for the verification of imperative recursive programs based on symbolic execution [11], forward reasoning [13] and functional semantics [14]: given a program together with its specification, forward symbolic execution analyzes the program top-down, checking that it is syntactically correct, each of the variable is initialized before it is used, and that each branch has a `Return` statement, and updates, eventually, the values of the variables or the path condition. A `Return` statement leaves the analysis of the program on the respective branch, generates a goal-oriented verification condition for it, and computes the expression of the output value(s) in terms of the input value(s) (functional semantics). The distinctive features of our approach are:

- All verification conditions (including termination condition) are formulated in the theory of the objects which are manipulated by the program.
- The notions of program, semantics, verification condition, and termination are precisely formalized in predicate logic.
- Termination is treated in a purely logical way, namely as the existence and uniqueness of the function implemented by the program.

The treatment of `While` loops has similarities to the treatment of recursive calls, namely, the loop can be compared to a recursive call of a new function together with the definition of it. The parameters of the new 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. Because they cannot be expressed by a composite term, fresh variables fulfilling certain conditions (the loop invariant and the negation of the loop condition) replace them.

Following these considerations, the verification conditions are generated using the following principles:

- For the program path which does not take into consideration the analysis of the current loop, the actual substitution is carried out, together with the negated loop condition, in the analysis of the program statements following the loop, respectively it is used for proving that the invariant holds at the beginning of the loop analysis.
- The symbolic values of the initialized variables when entering the loop must fulfill the loop invariant.
- The loop invariant is inductively preserved.
- The effect of the loop is encoded as intermediate assertions (loop invariant and negation of the loop condition) on the critical variables and used as accumulated assumptions for analyzing the rest of the program.
- The termination condition is an inductive principle, however still expressed as a first order formula by using a new (arbitrary but fixed) predicate symbol.

Related Work. Our approach follows the principles of symbolic execution, forward reasoning and functional semantics, but additionally gives formal definitions in a metatheory for the meta-level functions and predicate which characterize the object computation. To our knowledge, there is no other work which formalizes in predicate logic these static analysis methods. The definitions give the possibility of reflective view of the system by describing how the data (the state, the program, the verification conditions) are manipulated and by introducing a causal connection between the data and the status of computation.

However, the ideas from the formalization of the calculus are not completely new: [12] 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 [4] where the program equations (involving relation between current and post-state) are used to express nondeterminacy and termination. In the same manner, [15] 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], VDM [3].

We mention that our approach keeps the verification process very simple: the verification conditions generated are first order logic formulae, unlike [2] which uses dynamic logic, and the proof of correctness is kept at object-level without introducing a model of computation, unlike [16] which uses the fixpoint theory of programs.

Existing work on proving the *partial correctness* of symbolically executed programs relies on the Floyd's *inductive assertion method* [8]; the programs containing loops with unbounded number of iterations (`While` loops) and recursive calls, whose execution leads to infinite branches, are traversed inductively in order to generate the verification conditions. In the inductive traversal of the tree, additional assertions have to be provided, called *inductive assertions* or *invariants* in the case of `While` loops.

Existing work focuses on *termination* proofs of programs by finding termination terms [9] or by computing the closure of some well-founded relations [1]. Unlike this, we focus on the generation of the termination conditions similarly as it is done for partial correctness.

II. Preliminary Results

In the following we summarize the results of [7].

Our approach is purely logical. An *object theory* Υ is used for expressing the properties of the constants, functions and predicates which are used in the program. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) We consider the following types of functions:

- *basic* functions occur in the object theory; they only have input conditions, but no output conditions. Typical examples are the arithmetic operations in various number domains.
- *additional* functions occur in the object theory and are usually functions implemented by other programs. In the process of verification conditions generation only their specification will be used.

A *meta-theory* is further constructed for reasoning about the programs. It contains the properties of the meta-predicate Π – checks that a program is syntactical correct, that every branch contains the `Return` statement, and that each variable is initialized before it is used, and the meta-functions Σ – defines the semantics of a program, Γ – generates the verification conditions, and Θ – generates one termination condition. The output of the meta-functions consists in first order formulae which are to be understood as universally quantified over the free variables.

Predefined functions in the meta-theory are: *Vars* – returning the variables which occur in a term, $\langle ... \rangle$ – denoting tuples, and $\smile$ – denoting concatenation of tuples. The symbol $=$ is used as a logical equality both at meta- (in the definitions of the meta-level functions) and object level (e.g. as a predicate composing the condition of the `If` statement). The ascending precedence of the connectives in a formula is: $=$ (at meta-level), $\smile$, $\Rightarrow$, etc.

The programming constructs (statements), the program itself, as well as 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 imperative recursive language considered in our previous work was Turing complete: *abrupt statement* (`Return`), *assignments* (allowing *recursive calls*) and *conditionals* (`If` with one and two branches). *Recursive calls* are indicated by the presence of the function symbol f, conventionally corresponding to the function realized by the program. A program P is a tuple of statements, it takes as formal arguments a certain number of variables (denoted conventionally by $\overline{\alpha}$) and it returns a single value (denoted conventionally by β).

Our framework follows the paradigm of design by contract: we are reasoning about programs documented with pre- and postconditions, the first order logic formulae $I_f[\bar{\alpha}]$ and $O_f[\bar{\alpha}, \beta]$, respectively.

The algorithms are written in the Theorema procedural language [5], where `Program`, `Specification`, `Pre`, `Post` play the role of user interface commands.

For illustrating our approach we consider the algorithm computing the greatest common divisor (GCD) of two positive integers (*Example 1*). One could notice the incompleteness of the postcondition: a common divisor is outputted, not the greatest. It was simplified because of space reasons, but it has no influence on the symbolic execution of the program and is still interesting because is expressed by existential quantification.

The meta-predicate Π checks whether a program is syntactically correct, that each branch terminates with `Return` and that each variable is initialized before its usage.

For instance, in *Example 1* no syntax error is detected.

The semantics of a program is defined via the meta-level function Σ as being an implicit definition *at object level* of the function implemented by the program. It works by forward symbolic execution on all branches of the program, using as *state* the current substitution for the initialized

$$
\begin{array}{l}
\texttt{Specification}[\text{"GCD"}, G[\downarrow a, \downarrow b], \\
\quad \texttt{Pre} \rightarrow \texttt{a} \geq 0 \wedge \texttt{b} > 0, \\
\quad \texttt{Post} \rightarrow \underset{k1k2}{\exists\,\exists}((a = k1 * \beta) \wedge (b = k2 * \beta))] \\
\texttt{Program}[\text{"GCD"}, G[\downarrow a, \downarrow b], \\
\quad \texttt{If}[a = 0, \\
\quad\quad \texttt{Return}[b]]; \\
\quad \texttt{If}[b \neq 0, \\
\quad\quad \texttt{If}[a > b, \\
\quad\quad\quad a := G[a - b, b], \\
\quad\quad\quad a := G[a, b - a]]]; \\
\quad \texttt{Return}[a]], \\
\texttt{Specification} \rightarrow \texttt{Specification}[\text{"GCD"}]]
\end{array}
$$

Example 1. GCD Algorithm

variables, producing a formula with the following shape:

$$\{I_f[\bar{\alpha}] \Rightarrow \big(p_i[\bar{\alpha}] \Rightarrow (f[\bar{\alpha}] = g_i[\bar{\alpha}])\big) \mid i = 1..n\},$$

where f is a new (second order) symbol denoting conventionally the function defined by the program. In the case of recursive programs, f may occur in some p_i's (the accumulated `If` conditions on that path) and g_i's (the expression obtained by symbolic execution on the respective path).

Σ produces a conjunction of clauses, each clause representing a path of the program, a conditional definition for $f[\overline{\alpha}]$. Each clause depends on the accumulated [negated] conditions of the `If` statements leading to a certain `Return` statement, whose argument (symbolically evaluated) represents the corresponding value of $f[\overline{\alpha}]$. Note that Σ effectively translates the original program into a *functional* program. From this point on, one could reason about the program using the Scott fixpoint theory [13, p. 86], however we prefer a purely logical approach.

The semantics of the program implementing the GCD algorithm is as follows:

$$
\begin{array}{l}
a \geq 0 \wedge b > 0 \wedge (a = 0) \Rightarrow (G[a, b] = b) \\
a \geq 0 \wedge b > 0 \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \\
\quad \Rightarrow (G[a, b] = G[a - b, b]) \\
a \geq 0 \wedge b > 0 \wedge a \neq 0 \wedge (b = 0) \wedge a \leq b \\
\quad \Rightarrow (G[a, b] = G[a, b - a])
\end{array}
$$

The meta-function Γ produces the verification conditions as a conjunction of formulae at object level. It works similarly to Σ by using symbolic execution on all branches of the program, but additionally generates formulae using the following principles:

- coherence (safety) conditions: formulae with the shape $\Phi \Rightarrow I_h[\bar{t}]$, where I_h is the input condition of some function h, and $\bar{t}$ is a sequence of symbolic values of the function call. The formula Φ accumulates the conditions from `If` statements and the input and output conditions for the function calls on the respective branch;
- functional conditions: goal oriented formulae checking that the output condition with the currently returned value is a consequence of the accumulated conditions on the respective branch.

In most of the cases the programs contain calls of nested functions. When generating the verification conditions for the respective terms we ensure the safety conditions of all the functions of the call, by unwinding the term from the innermost to the outermost function symbol.

We implemented the function which generates the verification conditions in a prototype implementation `VCG`, integrated in Theorema, written in Mathematica [17]. It takes as input a program together with its specification, analyzes the program top-down and produces as output a set of first order formulae (i.e. verification conditions) constituting the input of a prover that checks their validity.
The verification conditions (simplified due to space reasons) for the GCD algorithm are generated with the command:

$$\texttt{VCG}[\texttt{Program}[\text{"GCD"}]]$$

$$a = 0 \wedge b > 0 \Rightarrow \underset{k1k2}{\exists\,\exists}\big((a = k1 * b) \wedge (b = k2 * b)\big) \tag{1}$$

$$a > 0 \wedge b > 0 \wedge a > b \Rightarrow a - b \geq 0 \wedge b > 0 \tag{2}$$

$$
\begin{array}{l}
a > 0 \wedge b > 0 \wedge a > b \wedge \underset{k1k2}{\exists\,\exists}\big((a - b = k1 * \mathrm{t1}) \wedge (b = k2 * \mathrm{t1})\big) \\
\quad \Rightarrow \underset{k1k2}{\exists\,\exists}\big((a = k1 * \mathrm{t1}) \wedge (b = k2 * \mathrm{t1})\big)
\end{array} \tag{3}
$$

$$a > 0 \wedge b > 0 \wedge \wedge a \leq b \Rightarrow a \geq 0 \wedge b - a > 0 \tag{4}$$

$$
\begin{array}{l}
a > 0 \wedge b > 0 \wedge b - a > 0 \wedge \underset{k1k2}{\exists\,\exists}((a = k1 * \mathrm{t2}) \wedge (b - a = k2 * \mathrm{t2})) \\
\quad \Rightarrow \underset{k1k2}{\exists\,\exists}((a = k1 * \mathrm{t2}) \wedge (b = k2 * \mathrm{t2}))
\end{array} \tag{5}
$$

$$a > 0 \wedge b > 0 \wedge b = 0 \Rightarrow \underset{k1k2}{\exists\,\exists}((a = k1 * a) \wedge (b = k2 * a)) \tag{6}$$

Remark 1:

1) The function G can not occur in the verification conditions thus its occurrences are replaced by the variables $t1$ and $t2$ (e.g. (3), (5)).
2) Additionally to the input variables, the newly introduced variables $k1$ and $k2$ are bound.
3) The program output has the expressions: b in (1), $t1$ in (3), $t2$ in (5), a in (6).

Programs with no recursive call always terminate. We consider in [7] the termination of imperative recursive programs expressed as: „The formula $\underset{\alpha}{\forall} I_f[\alpha] \Rightarrow O_f[\alpha, f[\alpha]]$ is a logical consequence of the object theory augmented with $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ (program semantics) is contradictory to Υ, which may happen when the program is recursive. Therefore, *it is crucial that the existence (and possibly the uniqueness) of an f satisfying $\Sigma[P]$ is a*

logical consequence of the object theory augmented with the verification conditions. More concretely, before using $\Sigma[P]$ as an assumption, one should guarantee that $\exists\limits_{f}\Sigma[P]$. The later is ensured by the termination condition which is expressed as an induction scheme developed from the structure of the recursion.

In our approach the termination condition is generated by the meta-function Θ, in which a new constant symbol π occurs, standing for an arbitrary predicate. It operates similarly to Γ by inspecting all the possible branches, collecting the conditions of the `If` statements and of the output specification of the additional functions occurring in the program, including the currently defined defined function f. In the last case, which happens for the recursive calls, one also collects the condition $\pi[\overline{\gamma}\sigma]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f.

For *Example 1* the termination condition is:

$$\underset{a\geq 0\wedge b>0}{\forall} \wedge \begin{cases} a=0 \Rightarrow \pi[a,b] \\ a\neq 0\wedge b\neq 0\wedge a>b\wedge \pi[a-b,b]\Rightarrow \pi[a,b] \\ a\neq 0\wedge b\neq 0\wedge a\leq b\wedge \pi[a,b-a]\Rightarrow \pi[a,b] \\ a\neq 0\wedge b=0\Rightarrow \pi[a,b] \end{cases}$$
$$\Rightarrow \underset{a\geq 0\wedge b>0}{\pi}[a,b]$$

III. A Calculus for Programs containing `While` Loops

In this work we extend the formal framework for analyzing the [total] correctness of `While` loops; it handles the partial correctness of abrupt terminating loops (Section III-C), but not their termination (Section III-D).

Notations: As *states* of execution we use substitutions σ (set of replacements of the form $\{var \rightarrow expr\}$). Note that we sometimes write $\{\overline{var} \rightarrow \overline{expr}\}$ instead of $\{var_1 \rightarrow expr_1, var_2 \rightarrow expr_2, ...\}$.

All the formulae composing the meta-definitions are to be understood as universally quantified over the meta-variables of various types as follows; $v \in \mathcal{V}$ is a variable from the set $\mathcal{V}$ of variables, $V \subset \mathcal{V}$ is the set of initialized variables, φ is an object level formula standing for the `While` condition, B is a tuple of statements representing the loop body. The symbol ι denotes conventionally the loop invariant and it is a conjunction of first order logical formulae. We call the variables $\overline{\delta}$ modified within the `While` loop critical variables; they are replaced with new symbolic values $\overline{\delta_0}$ when entering the loop and with $\overline{\delta'}$ when exiting the loop body.

In conjunction with the critical variables there are two special substitutions used in the calculus: $\sigma_0 = \{\overline{\delta} \rightarrow \overline{\delta_0}\}\sigma$ appears each time program analysis with fresh symbols for the critical variables has to be performed (e.g. the analysis of the `While` loop starts with this type of substitution) and $\sigma' = \{\overline{\delta} \rightarrow \overline{\delta'}\}\sigma$ – when exiting the `While` loop the program analysis continues with the values for the critical variables at loop exit, denoted by the symbolic values $\overline{\delta'}$.

We consider the term t and γ, $\overline{\gamma}$ – a variable and/or a constant and respectively a sequence of variables and/or constants from the object theory Υ.

Loops are meta-terms with the structure: `While`$[\varphi, \iota, B]$.

Example 2 is the Dijkstra algorithm for computing simultaneously the greatest common divisor (GCD) and the least common multiple (LCM) of two positive integers. Note that the functions LCM and GCD associated to the corresponding algorithms are in the object theory.

`Specification`["`GCD − LCM`", $LG[\downarrow a, \downarrow b]$,
  `Pre` $\rightarrow a>0\wedge b>0$,
  `Post` $\rightarrow \beta = LCM(a,b)$]
`Program`["`GCD − LCM`", $LG[\downarrow a, \downarrow b]$,
  $x=a; y=b; u=b; v=a;$
  `While`$[x\neq y,$
    $GCD[x,y]=GCD[a,b]\wedge x>0\wedge y>0$
      $\wedge\, x*u+y*v=2*a*b,$
    `If`$[x>y,$
      $x=x-y; v=v+u,$
      $y=y-x; u=u+v]];$
  `Return`$[(u+v)/2]]$,
`Specification` $\rightarrow$ `Specification`["`GCD − LCM`"]]

Example 2. LCM-GCD Algorithm

Example 3 is the linear algorithm for searching a particular value in an array.

`Specification`["`LinSearch`", $LS[\downarrow A, \downarrow n, \downarrow e]$,
  `Pre` $\rightarrow n\geq 1$,
  `Post` $\rightarrow \underset{1\leq k\leq n}{\exists} A_{[k]}=e \Rightarrow A_{[\beta]}=e$
    $\wedge \underset{1\leq k\leq n}{\forall} A_{[k]}\neq e \Rightarrow \beta=0$
`Program`["`LinSearch`", $LS[\downarrow A, \downarrow n, \downarrow e]$,
  $i=1;$
  `While`$[i\leq n,$
    $(i\leq n+1)\Rightarrow \underset{1\leq k<i}{\forall} A_{[k]}\neq e,$
    `If`$[e=A_{[i]},$
      `Return`$[i]];$
    $i=i+1];$
  `Return`$[0]]$,
`Specification` $\rightarrow$ `Specification`["`LinSearch`"]]

Example 3. Linear Search Algorithm

A. Syntax

The predicate Π checks a program for syntactic correctness including the fact that each variable is initialized and that each branch contains a `Return` statement.

Definition 2:

1) $\Pi[P] \Leftrightarrow \Pi[\{\bar{\alpha}\}, P]$

2) $\Pi[V, \langle \texttt{Return}[t]\rangle \backsim P] \Leftrightarrow Vars[t] \subseteq V$

3) $\Pi[V, \langle v\colon = t\rangle \backsim P] \Leftrightarrow \bigwedge \begin{cases} Vars[t] \subseteq V \\ \Pi[V \cup \{v\}, P] \end{cases}$

4) $\Pi[V, \langle \texttt{If}[\varphi, P_T, P_F]\rangle \backsim P] \Leftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ \Pi[V, P_T \backsim P] \\ \Pi[V, P_F \backsim P] \end{cases}$

5) $\Pi\big[V, \langle \texttt{While}[\varphi,\ \iota,\ B]\rangle \backsim P\big] \Leftrightarrow$
$\bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ \Pi\big[V, B \backsim \langle \texttt{Return}[True]\rangle\big] \\ \Pi\big[V, P\big] \end{cases}$

6) $\Pi[V, P] = \mathbb{F}$

In the case of `While` loops (Definition 2.5), the `Return` statement may occur in the loop body B, but mandatory in P .

B. Semantics

The formula standing for the program semantics is generated using the following inductive definitions:

Definition 3:

1) $\Sigma[P] = \big(I_f[\bar{\alpha}] \Rightarrow \Sigma[\{\bar{\alpha} \to \bar{\alpha}_0\},\ P]\{\bar{\alpha_0} \to \bar{\alpha}\}\big)$

2) $\Sigma[\sigma,\ \langle \texttt{Return}[t]\rangle \backsim P] = (f[\bar{\alpha}_0] = t\sigma)$

3) $\Sigma[\sigma,\ \langle v := t\rangle \backsim P] = \Sigma[\sigma\{v \to t\sigma\},\ P]$

4) $\Sigma[\sigma,\ \langle \texttt{If}[\varphi,\ P_T,\ P_F]\rangle \backsim P] =$
$\bigwedge \begin{cases} \varphi\sigma \Rightarrow \Sigma[\sigma,\ P_T \backsim P] \\ \neg\varphi\sigma \Rightarrow \Sigma[\sigma,\ P_F \backsim P] \end{cases}$

5) $\Sigma\big[\sigma,\ \langle\rangle\big] = True$

6) $\Sigma\big[\sigma,\ \langle \texttt{While}[\varphi,\ \iota,\ B]\rangle \backsim P\big] =$
$\bigwedge \begin{cases} \neg\varphi\sigma \Rightarrow\ \Sigma[\sigma,\ P] \\ \big(\varphi\sigma_0 \wedge \iota\sigma_0 \Rightarrow \Sigma\big[\sigma_0,\ B\big]\big)\{\bar{\delta}_0 \to \bar{\delta}\} \\ \neg\varphi\sigma' \wedge \iota\sigma' \ \Rightarrow\ \Sigma\big[\sigma',\ P\big] \end{cases}$

For `While` loops the program semantics is constructed by considering three possible execution paths: the loop is not executed at all or at least once. In the first case the program semantics is constructed by considering the current substitution and the statements after the loop body. In the second case distinction between the occurrence and absence of the abrupt statement `Return` in the loop body is made: if it occurs in the loop body, then the program function is computed on the respective branch with the current symbolic values for the critical variables, otherwise, the loop characterization (the negation of the loop condition and the loop invariant), used as accumulated assumptions, and the substitution at the exit point of the loop – σ' – are used for further computing the program function on the respective path. Note that Definition 3.5 handles the case when the loop analysis reaches its end and no abrupt statement was encountered: the loop does not bring new information in the semantics of the program.

The program semantics for *Example 2* is:

$$a > 0 \wedge b > 0 \wedge (a = b) \Rightarrow LG[a, b] = b \tag{7}$$

$$a > 0 \wedge b > 0 \Rightarrow True \tag{8}$$

$$a > 0 \wedge b > 0 \Rightarrow True \tag{9}$$

$$a > 0 \wedge b > 0 \wedge x' = y' \wedge GCD[x', y'] = GCD[a, b] \wedge x' > 0 \wedge y' > 0 \wedge x' * u' + y' * v' = 2 * a * b \Rightarrow LG[a, b] = \frac{u' + v'}{2} \tag{10}$$

but actually only (7) and (10) are relevant.

For the algorithm performing linear search in an array (*Example 3*), the program semantics is as follows:

$$n \geq 1 \wedge (1 > n) \Rightarrow LS[A, n, e] = 0 \tag{11}$$

$$n \geq 1 \wedge i \leq n \wedge (i \leq n + 1 \Rightarrow \underset{1 \leq k < i}{\forall} A_{[k]} \neq e) \wedge e = A_{[i]} \Rightarrow LS[A, n, e] = i \tag{12}$$

$$n \geq 1 \wedge i \leq n \wedge (i \leq n + 1 \Rightarrow \underset{1 \leq k < i}{\forall} A_{[k]} \neq e) \wedge e \neq A_{[i]} \Rightarrow True \tag{13}$$

$$n \geq 1 \wedge i' > n \wedge (i' \leq n + 1 \Rightarrow \underset{1 \leq k < i'}{\forall} A_{[k]} \neq e) \tag{14}$$

$$\Rightarrow LS[A, n, e] = 0 \tag{15}$$

Note that the value of LS can not be 0, because assumptions of the formula (11) are contradictory, and that (13) does not contribute to the program semantics.

C. Partial Correctness

As we mentioned before, the meta-level function Γ generates *coherence (safety)* conditions and *functional* conditions, which together ensure *partial correctness*.

Definition 4:

1) $\Gamma[P] = \Gamma[\{\bar{\alpha} \to \bar{\alpha}_0\},\ I_f[\bar{\alpha}_0],\ P]\{\bar{\alpha}_0 \to \bar{\alpha}\}$

2) $\Gamma[\sigma,\ \Phi,\ \langle \texttt{Return}[\gamma]\rangle \backsim P] = \big\langle \Phi \Rightarrow O_f[\bar{\alpha}_0, \gamma\sigma]\big\rangle$

3) $\Gamma[\sigma,\ \Phi,\ \langle v := \gamma\rangle \backsim P] = \Gamma[\sigma\{v \to \gamma\sigma\},\ \Phi,\ P]$

4) $\Gamma[\sigma,\ \Phi,\ \langle v := h[\overline{\gamma}]\rangle \backsim P] =$
$\big\langle \Phi \Rightarrow I_h[\overline{\gamma}\sigma]\big\rangle \backsim \Gamma[\sigma\{v \to h[\overline{\gamma}\sigma]\},\ \Phi \wedge I_h[\overline{\gamma}\sigma],\ P]$

5) $\Gamma[\sigma,\ \Phi,\ \langle v := g[\overline{\gamma}]\rangle \backsim P] = \big\langle \Phi \Rightarrow I_g[\overline{\gamma}\sigma]\big\rangle$
$\backsim \Gamma[\sigma\{v \to c\}, \Phi \wedge I_g[\overline{\gamma}\sigma] \wedge O_g[\overline{\gamma}\sigma, c],\ P]$

6) $\Gamma[\sigma,\ \Phi,\ \langle \texttt{If}[\varphi,\ P_T,\ P_F]\rangle \backsim P] =$
$\Gamma[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \backsim P] \backsim \Gamma[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \backsim P]$

7) $\Gamma\big[\sigma,\ \Phi,\ \langle\rangle\big] = \big\langle \Phi \Rightarrow \iota\sigma\big\rangle$

8) $\Gamma\big[\sigma,\ \Phi,\ \langle \texttt{While}[\varphi,\ \iota,\ B]\rangle \backsim P\big] =$
$\backsim \begin{cases} \Gamma\big[\sigma, \Phi \wedge \neg\varphi\sigma, P\big] \\ \big\langle \Phi \Rightarrow \iota\sigma\big\rangle \\ \big(\Gamma\big[\sigma_0, \Phi \wedge \iota\sigma_0 \wedge \varphi\sigma_0, B\big]\big)\{\bar{\delta_0} \to \bar{\delta}\} \\ \Gamma\big[\sigma', \Phi \wedge \iota\sigma' \wedge \neg\varphi\sigma', P\big] \end{cases}$

The verification conditions for the `While` loop are generated as follows: at its entry point the invariant must hold, the invariant must be inductively preserved by the operations

done in the loop body, the loop invariant and the negated loop condition are used as premises for further analyzing the program. As we said before, the loop is viewed as a separate module, so the analysis of the loop body is done with new fresh values for the critical variables. The values for the critical variables which are computed by the loop are encoded in the primed values introduced and they are used further in the analysis of the program.

The set of verification conditions for *Example 2* is:

$$a > 0 \wedge b > 0 \wedge (a = b) \Rightarrow (LCM(a,b) = \frac{b+a}{2})$$

$$a > 0 \wedge b > 0 \Rightarrow \big(GCD[a,b] = GCD[a,b] \wedge a > 0 \wedge b > 0 \wedge a*b+b*a = 2*a*b\big)$$

$$\big(a > 0 \wedge b > 0 \wedge GCD[x,y] = GCD[a,b] \wedge x > 0 \wedge y > 0 \wedge x*u+y*v = 2*a*b \wedge x \neq y \wedge x > y\big)$$
$$\Rightarrow \big(GCD[x-y,y] = GCD[a,b] \wedge x-y > 0 \wedge y > 0 \wedge (x-y)*u+y*(v+u) = 2*a*b\big)$$

$$\big(a > 0 \wedge b > 0 \wedge GCD[x,y] = GCD[a,b] \wedge x > 0 \wedge y > 0 \wedge x*u+y*v = 2*a*b \wedge x \neq y \wedge x \leq y\big)$$
$$\Rightarrow \big(GCD[x,y-x] = GCD[a,b] \wedge x > 0 \wedge y-x > 0 \wedge x*(u+v)+(y-x)*v = 2*a*b\big)$$

$$\big(a > 0 \wedge b > 0 \wedge GCD[x',y'] = GCD[a,b] \wedge x' > 0 \wedge y' > 0 \wedge x'*u'+y'*v' = 2*a*b \wedge x' = y'\big)$$
$$\Rightarrow \big(\frac{u'+v'}{2} = LCM[a,b]\big)$$

For *Example 3* the verification conditions are:

$$n \geq 1 \wedge 1 > n$$
$$\Rightarrow \big(\underset{1\leq k\leq n}{\exists} A_{[k]} = e \Rightarrow A_{[0]} = e \wedge \underset{1\leq k\leq n}{\forall} A_{[k]} \neq e \Rightarrow 0 = 0\big)$$

$$n \geq 1 \wedge i \leq n \wedge (i \leq n+1 \Rightarrow \underset{1\leq k<i}{\forall} A_{[k]} \neq e) \wedge e = A_{[i]}$$
$$\Rightarrow \big(\underset{1\leq k\leq n}{\exists} A_{[k]} = e \Rightarrow A_{[i]} = e \wedge \underset{1\leq k\leq n}{\forall} A_{[k]} \neq e \Rightarrow i = 0\big)$$

$$n \geq 1 \wedge i \leq n \wedge (i \leq n+1 \Rightarrow \underset{1\leq k<i}{\forall} A_{[k]} \neq e) \wedge e \neq A_{[i]}$$
$$\Rightarrow \big(i \leq n \Rightarrow \underset{1\leq k\leq i}{\forall} A_{[k]} \neq e\big)$$

$$n \geq 1 \wedge i' > n \wedge (i' \leq n+1 \Rightarrow \underset{1\leq k<i'}{\forall} A_{[k]} \neq e) \Rightarrow$$
$$\big(\underset{1\leq k\leq n}{\exists} A_{[k]} = e \Rightarrow A_{[i']} = e \wedge \underset{1\leq k\leq n}{\forall} A_{[k]} \neq e \Rightarrow i' = 0\big)$$

D. Termination

We present a novel method for handling the termination of `While` loops: a loop is viewed as a separate module whose template termination condition is expressed as an induction principle depending on the structure of the loop, where a new predicate symbol π is used. The termination condition is generated by the meta-level function Θ, which is extended to handle termination of imperative non-recursive programs containing normal terminating non-nested `While` loops.

In the termination condition the crucial role is played by the critical variables and therefore Θ has to remember which critical variables correspond to each loop in order to attribute the right arguments to the predicate π. If for *non-nested* `While` *loops programs* they can be determined and distinguished easily by analyzing the assignments from each loop body, for *nested* `While` *loops programs* (including the abrupt terminating ones) the critical variables of the corresponding loop have to be determined. Moreover Θ has to distinguish between abrupt and, respectively, normal terminating loops.

In the following sections we present the termination of programs that have non-nested, normal terminating, `While` loops.

1) Termination of the `While` *Loops:* We approach the termination of the `While` loops by placing cuts at the entry and exit points into the source code of the program. In this manner the loop is viewed as a separate module whose template termination condition formula is:

$$\underset{\bar{\delta}:\Phi}{\forall} \wedge \left\{\begin{array}{l}\neg\varphi \Rightarrow \pi[\bar{\delta}] \\ \varphi \wedge \Psi \wedge \pi[\bar{\delta}\sigma_W\{\bar{\delta_0} \rightarrow \bar{\delta}\}] \Rightarrow \pi[\bar{\delta}]\end{array}\right\} \Rightarrow \pi[\bar{\delta}\sigma_I]$$

The template termination condition states the following: for all critical variables satisfying the accumulated path condition Φ at the entry point of the loop, if the loop condition is not fulfilled then the loop terminates for the corresponding critical variables and if the loop analysis leads to loop termination for the same critical variables then the respective loop terminates for the input values of the critical variables. The loop analysis implies collecting in Ψ the loop invariant, the conditions of the `If` statements, if it is the case, and the output characterization of the additional functions encountered, and also formulae of the type $\pi[\bar{\delta}\sigma_W\{\bar{\delta_0} \rightarrow \bar{\delta}\}]$ – that is the arbitrary predicate applied to the current symbolic values $\sigma_W\{\bar{\delta_0} \rightarrow \bar{\delta}\}$ of the critical variables of the `While` loop. This is done for each possible execution path of the loop.

Consider the loop of *Example 2*. The corresponding termination condition is:

$$\underset{\substack{x,y,u,v \\ a>0\wedge b>0}}{\forall} \wedge \left\{\begin{array}{l}(x = y) \Rightarrow \pi[x,y,u,v] \\ (x > y \wedge GCD[x,y] = GCD[a,b] \\ \quad \wedge x > 0 \wedge y > 0 \wedge x*u+y*v = 2*a*b \\ \quad \wedge \pi[x-y,y,u,v+u]) \Rightarrow \pi[x,y,u,v] \\ (x < y \wedge GCD[x,y] = GCD[a,b] \\ \quad \wedge x > 0 \wedge y > 0 \wedge x*u+y*v = 2*a*b \\ \quad \wedge \pi[x,y-x,u+v,v]) \Rightarrow \pi[x,y,u,v]\end{array}\right.$$
$$\Rightarrow \pi[a,b,b,a]$$

The termination analysis of the loop stops if it finds abrupt statements. If the loop terminates due to `break` then a termination condition of the respective loop on the current path is generated and the control is transferred to the statement immediately following that loop.

If the loop terminates due to `Return` statement then the program terminates on the respective execution path and consequently termination conditions for each loop analyzed until the respective point have to be generated. The calculus dealing with these situations is under development.

2) Termination of Non-Recursive Programs containing Normal Terminating Non-Nested `While` *Loops:* The general idea behind termination of mentioned programs is to generate a tuple of termination conditions corresponding to the `While` loops of the program. They ensure the termination of the program itself. For simplicity of presentation we assume that `Return`, assignments, `If` and `While` conditions do not contain composite terms. A presentation of the definitions corresponding to recursive calls and additional functions can be found in [7].

The inductive definitions for the meta-function Θ are as follows.

Definition 5:

1) $\Theta[P] = \Theta[\{\bar{\alpha} \to \bar{\alpha_0}\},\ I_f[\bar{\alpha}],\ P]\{\bar{\alpha}_0 \to \bar{\alpha}\}$
2) $\Theta[\sigma,\ \Phi,\ \langle \texttt{Return}[\bar{\gamma}]\rangle \smile P] = \langle\rangle$
3) $\Theta[\sigma,\ \Phi,\ \langle v := \gamma\rangle \smile P] = \Theta[\sigma\{v \to \gamma\sigma\},\ \Phi,\ P]$
4) $\Theta[\sigma,\ \Phi,\ \langle v := h[\bar{\gamma}]\rangle \smile P] = \Theta[\sigma\{v \to h[\bar{\gamma}\sigma]\},\ \Phi,\ P]$
5) $\Theta[\sigma,\ \Phi,\ \langle \texttt{If}[\varphi,\ P_T,\ P_F]\rangle \smile P] =$
   $\Theta[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \smile P] \wedge \Theta[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \smile P]$
6) $\Theta\big[\sigma,\ \Phi,\ \langle\rangle\big] = \big((\Phi \wedge \pi[\bar{\delta}\sigma]) \Rightarrow \pi[\bar{\delta}]\big)$
7) $\Theta[\sigma,\ \Phi,\ \langle \texttt{While}[\varphi,\ \iota,\ B]\smile P] =$
$$\smile \begin{cases} \Theta\big[\sigma, \Phi \wedge \neg\varphi\sigma, P\big] \\ \Big\langle \underset{\bar{\delta}:\Phi}{\forall\wedge} \begin{cases} \big(\neg\varphi\sigma_0 \Rightarrow \pi[\bar{\delta}\sigma_0]\big)\{\bar{\delta}_0 \to \bar{\delta}\} \\ \Theta\big[\sigma_0, \varphi\sigma_0 \wedge \iota\sigma_0, B\big]\{\bar{\delta}_0 \to \bar{\delta}\} \end{cases} \Rightarrow \pi[\bar{\delta}\sigma_I] \Big\rangle \\ \Theta\big[\sigma', \Phi \wedge \iota\sigma' \wedge \neg\varphi\sigma',\ P\big] \end{cases}$$

The termination analysis of the program P starts with symbolic values for the input variables and it is performed only for those satisfying the input condition (Definition 5.1). The meta-function Θ works by symbolic execution updating eventually the substitution (e.g. Definitions 5.3, 5.4, or the substitution at the entry and exit of the loop) and the conditional part of the termination condition (e.g. Definitions 5.5, 5.6) or generating the termination condition of the whole program. (e.g. Definition 5.2). Each time Definition 5.7 is applied, a new symbol π is generated for expressing the termination condition of the respective loop.

End of the loop requires the generation of the termination condition on the respective path, case when the symbol π associated with the currently analyzed `While` loop has to be used. If the loop terminates then its characterization together with the accumulated assumptions until the loop entry are used as a conditional formula for reasoning about the termination rest of the program. The loop characterization is a formula expressing the relationship between the critical variables and some outer loop variables – actually the loop invariant, as well as the loop condition falsified by the exit values of the loop.

For the *Example 2* there were three paths taken into consideration, but only the one analyzing the loop counts to the termination analysis of the program.

$$\underset{\substack{x,y,u,v \\ a>0\wedge b>0}}{\forall} \wedge \begin{cases} (x = y) \Rightarrow \pi[x, y, u, v] \\ (x > y \wedge GCD[x, y] = GCD[a, b] \\ \quad \wedge x > 0 \wedge y > 0 \wedge x * u + y * v = 2 * a * b \\ \quad \wedge \pi[x - y, y, u, v + u]) \Rightarrow \pi[x, y, u, v] \\ (x < y \wedge GCD[x, y] = GCD[a, b] \\ \quad \wedge x > 0 \wedge y > 0 \wedge x * u + y * v = 2 * a * b \\ \quad \wedge \pi[x, y - x, u + v, v]) \Rightarrow \pi[x, y, u, v] \end{cases}$$
$$\Rightarrow \pi[a, b, b, a]$$

The Binary Division algorithm (*Example 4*) [10] contains two non-nested `While` loops. (The integer type of the variable k was omitted for the simplicity of the presentation.)

```
Specification["BinaryDivision", BD[↓ A, ↓ B],
  Pre → A ≥ 0 ∧ B > 0,
  Post → β = A%B]
Program["BinaryDivision", BD[↓ A, ↓ B],
  q = 0; r = A; b = B;
  While[r ≥ b,
    (∃_{k≥0} b = 2^k * B) ∧ q = 0 ∧ r = A ∧ A ≥ 0 ∧ B > 0,
    b = 2 * b];
  While[b ≠ B,
    A = q*b+r ∧ r ≥ 0 ∧ (∃_{k≥0} b = 2^k * B) ∧ B > 0 ∧ b > r,
    q = 2 * q; b = b/2;
    If[r ≥ b,
      q = q + 1; r = r − b]];
  Return[r]]
Specification → Specification["BinaryDivision"]]
```

Example 4. Binary Division Algorithm, by Kaldewaij

It terminates if the below termination conditions are valid.

$$\underset{\substack{b,A,B \\ A\geq 0\wedge B>0}}{\forall} \wedge \begin{cases} (r < b) \Rightarrow \pi_1[b] \\ \big(r \geq b \wedge \big(\underset{k\geq 0}{\exists} b = 2^k * B\big) \wedge q = 0 \wedge r = A \\ \quad \wedge A \geq 0 \wedge B > 0 \wedge \pi_1[2 * b]\big) \Rightarrow \pi_1[b] \end{cases}$$
$$\Rightarrow \pi_1[B]$$

$$\underset{\substack{q,b,r,A,B \\ A\geq 0\wedge B>0 \\ A<B}}{\forall} \wedge \begin{cases} (b = B) \Rightarrow \pi_2[q, b, r] \\ \big(b \neq B \wedge A = q * b + r \wedge r \geq 0 \\ \quad \wedge \big(\underset{k\geq 0}{\exists} b = 2^k * B\big) \wedge B > 0 \wedge b > r \wedge r \geq b \\ \quad \wedge \pi_2[2 * q + 1, b/2, r - b]\big) \Rightarrow \pi_2[q, b, r] \\ \big(b \neq B \wedge A = q * b + r \wedge r \geq 0 \\ \quad \wedge \big(\underset{k\geq 0}{\exists} b = 2^k * B\big) \wedge B > 0 \wedge b > r \wedge r < b \\ \quad \wedge \pi_2[2 * q, b/2, r]\big) \Rightarrow \pi_2[q, b, r] \end{cases}$$
$$\Rightarrow \pi_2[0, B, A]$$

$$\forall_{\substack{q,b,r,A,B\\ A\geq 0\wedge B>0\\ r=A\wedge A\geq 0\\ B>0}} \exists_{k\geq 0} b' = 2^k * B \wedge q = 0 \wedge r < b' \wedge$$

$$\left\{\begin{array}{l} (b = B) \Rightarrow \pi_2[q, b, r] \\ b \neq B \wedge A = q * b + r \wedge r \geq 0 \\ \quad \wedge \big(\exists_{k\geq 0} b = 2^k * B\big) \wedge B > 0 \wedge b > r \\ \quad \wedge r \geq b \wedge \pi_2[2 * q + 1, b/2, r - b] \\ \quad \Rightarrow \pi_2[q, b, r] \\ \big(b \neq B \wedge A = q * b + r \wedge r \geq 0 \\ \quad \wedge \big(\exists_{k\geq 0} b = 2^k * B\big) \wedge B > 0 \wedge b > r \\ \quad \wedge r < b \wedge \pi_2[2 * q, b/2, r]\big) \Rightarrow \pi_2[q, b, r] \end{array}\right\} \Rightarrow \pi_2[0, b', A]$$

Actually there are seven paths to be considered for the termination of the program, but the ones which do not take into consideration the analysis of loops at all, the ones which consider that all the loops terminates, and the ones obtained by combining the previous two situations do not lead to termination conditions.

IV. Conclusion

The calculus presented in this paper combines forward symbolic execution and functional semantics for reasoning about imperative non-recursive programs.

A main direction for future work consists in the extension of the calculus to abrupt terminating [nested] loops. Moreover one needs to develop efficient methods for proving the verification conditions.

Acknowledgment

The first author was supported by the Upper Austrian Government.

References

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

[2] B. Beckert, R. Hähnle, and P. Schmitt (eds.), *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, Springer-Verlag, 2007.

[3] D. Bjørner and M. Henson, *Logics of Specification Languages*, Springer, 2008.

[4] 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.

[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 (english).

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

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

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

[10] A. Kaldewaij, *Programming: the Derivation of Algorithms*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[11] J. King, *A New Approach to Program Testing*, Proceedings of the international conference on Reliable software (New York, NY, USA), ACM, 1975, pp. 228–233.

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

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

[14] 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.

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

[16] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, USA, 1981.

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