Soundness of a Logic-Based Verification Method for Imperative Loops

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

***Abstract*—We present a logic-based verification method for imperative loops (including ones with abrupt termination) and the automatic proof of its soundness. The verification method consists in generating verification conditions for total correctness of an imperative loop annotated with an invariant. We realized, in the *Theorema* system (www.theorema.org), the automatic proof of the soundness of verification method: if the verification conditions hold, then the imperative loop is totally correct with respect to its given invariant. The approach is simpler than the others because it is based on functional semantics (no additional theory of program execution is necessary) and produces verification conditions in the object theory of the program. The computer-supported proofs reveal the minimal collection of logical assumptions (some from natural number theory) and logical inferences (including induction) which are necessary for the soundness of the verification technique.**

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

I. Introduction

The soundness and relative completeness proofs of the classical Hoare logic are well-established for sequential imperative programming languages. The same holds for logics extending Hoare logic with recursive calls, abrupt termination, exceptions, object-oriented features, etc. [10], [1], [22], [29]. These proofs are mainly done by defining the semantics in type theory [29] and by using the proof assistants Coq [3], Isabelle/HOL [21], PVS [24] in the (interactive) proofs. In a functional setting, the most common way of defining the semantics of the programs is to use Scott fixed-point theory [28].

Additionally to the definition of semantics, these proofs require to define the notion of termination. For imperative languages:

1) if the semantics defines a memory model of the program then inference rules for both partial and termination are introduced,
2) if an axiomatic semantics is defined then inference rules for termination are defined only for iterative structures (`while` loops); these inference rules capture the well-foundedness property of the iterative structure.

In the functional setting, termination is defined as being the least fix-point of certain recursive operators.

Needless to say, the complexity of the proofs of the program logics depends on the choice of semantics and definition of termination. The complexity plays a crucial role especially if one aims at the automatization of such proofs.

The present paper tries to avoid complex proofs by defining semantics and termination in the same logic as the one of the program. It focusses on the automatic proof of soundness, in the *Theorema* system [7], of a method handling abruptly terminating `while` loops in imperative programs. The method is based on forward symbolic execution [13] and functional semantics [19]. Our main aim is 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 has the potential to increase the confidence in program verification tools and even to reveal some foundational relations between logic and programming.

The distinctive features of our approach are:

- Program correctness, and implicitly the loop correctness, is expressed in predicate logic, without using any additional theoretical model for program semantics or program execution, but only using the so-called object theories, theories relevant to the predicates, constants and functions present in the program text.
- The semantics of a loop is the implicit definition, at object level, of the function implemented by the loop.
- Termination is defined as an induction principle developed from the structure of the program with respect to `while` loops.

The correctness statement of `while` loops states that the loop invariant is inductively preserved by the application of the loop semantics.

For the soundness proofs, the entire knowledge base is formulated only in the logic on which the program operates except some axioms of natural number theory (including induction over natural numbers). Moreover, the proofs are performed using mainly first-order inferences (exception is Skolemization). Note that, although the proof is automatic, the effort of performing it is comparable to the ones performed by the interactive theorem provers. However, the benefit is that all the assumptions we made appear explicitly in the knowledge base and therefore the theory is easier to explore in order to detect inconsistencies.

We identified a reasonable-size knowledge base and a small set of inference rules which are handled efficiently during

proof search by our predicate logic prover implemented in the *Theorema* system.

Our 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). Finally, the formal specification and the verification of the method are performed in the same framework, namely *Theorema* system. This facilitates reasoning at object and meta-level in the same system.

Related Work. Our approach follows the principles of forward symbolic execution [13] and functional semantics [19], but additionally gives formal definitions in a metatheory 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 [4] where the program equations (involving relation between current and post-state) are used to express nondeterminacy and termination. In the same manner, [26] presents the formal calculus for imperative languages containing complex structures. Specification languages used in the framework of verification tools also use this concept – see e.g. JML [16].

Nowadays, symbolic execution [13] is widely used in conjunction with other verification techniques [11].

Program logics for reasoning about programs with abrupt statements are implemented in many state-of-the-art program verifiers.

In the KeY system [1], which has a first-order predicate programming logic (called Java Card DL) with subtyping extended with parameterized modal operators, there are two ways of handling abrupt termination of `while` loops: *1*) syntactically, by enriching the logic with labeled modalities $[]_R$ and $\langle\rangle_R$, referring to the reason R of possible abrupt termination; *2*) semantically, by transforming the program into an equivalent one which catches all top-level exceptions and thus always terminates normally. Parts of the programming logic were proven correct using an existing Isabelle/HOL formalization [29] of Java semantics.

We prove the soundness of the method for loops with abrupt termination by transforming the loop into a normal terminating one (Section III-B). The transformation method looks similar to the one performed by KeY, however, we did not find any references on how the translation is performed, nor how the proof of the correctness was handled. Distinct to KeY system, our programming logic is first-order logic extended with meta-level constructs representing the program statements.

In ESC/JAVA 2 [8], abruptly terminating loops can be modeled by `throw`/`catch` clauses. However, the system uses an unsound calculus; one source of unsoundness is the loop-unrolling mechanism rather than a loop invariant. Unlike this, our method is sound because we are using loop invariants to characterize loops.

In the LOOP tool [2], program semantics is described in type theory and has a memory model and semantics inheritance as basic ingredients. A Hoare-like calculus including abrupt termination is developed which is proved to be sound w.r.t. their approach to semantics. The correctness proofs of the calculus are done interactively in PVS [24] and Isabelle/HOL [21]. On the contrary, our approach does not need any memory model because we are working in a functional environment and the proofs are done automatically in the *Theorema* system.

The authors of [25] develop a structural operational semantics and Hoare-like logic as part of the Jive system [20]. The program logic is interactively proved sound w.r.t. the semantics by translating both of them in higher-order logic using Isabelle/HOL [21]. In comparison with the LOOP system, which reasons at semantic level, the reasoning of JIVE is at syntactic level. Contrary to this, our semantics is expressed in the logic on which the program operates and the correctness proof of the calculus as well, except some second order inferences.

A formalized semantics (in higher-order logic) of the C programming language is given in [22]. It handles the cases of abrupt termination by translating, at syntactic level, the abruptly terminating program into a normal terminating one and deriving post-conditions for each case of termination. The formalization is done in Isabelle/HOL [21].

The idea of using induction for termination proving has been widely used explicitly [5] or implicitly in the form of well-founded induction [23]. These proving techniques can be seen in the context of our work as methods for proving certain classes of inductive termination conditions that we generate. Note that in our approach termination is formulated as an induction principle and not used as a proving technique for termination as in the existing approaches.

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 [21], HOL4 [9] and Coq [3] are basically using the recursion package TFL [27] 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[1] 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

[1]A critical variable is a program variable modified in the loop body.

termination proofs, like e. g. termination of term rewriting systems http://www.termination-portal.org/, the size-change termination principle [17] or the approaches for proving the termination of industrial-size code (Microsoft Windows Operating System Drivers) [5].

In order to prove the correctness of the `while` loops in the classical Hoare logic it suffices to prove that the invariant holds upon loop execution and that a termination term exists. In case of abrupt terminating loops, one way is to introduce, at syntactic level, the notion of abnormal state [10] in the correctness statement. In these approaches the correctness proof was done by proving the correctness of the loop depending on the current statement which can occur in the loop body, therefore logical formulae and program statements appear in the proof. In our approach, we transform the loop into logical formulae and prove loop correctness based on them. Therefore the computed-supported proof in our case is simpler because it has to deal only with formulae from the logic on which the program operates.

II. Logical Foundations

In this section we introduce the program model and exemplify it on an example. Algorithm 1 presents a searching

Algorithm 1 Linear Search

1. `in` a: `array of integers`;
   n, e: `integers` where $n \geq 0$
2. `out` β: `boolean` where
   $\Big(\underset{0\leq j<n}{\forall} a[j] \neq e \wedge \beta = \mathbb{F}\Big) \vee \Big(\underset{0\leq j<n}{\exists} a[j] = e \wedge \beta = \mathbb{T}\Big)$
3. $i := 0$; $y := \mathbb{F}$;
4. `while` $(i < n)$ `do`
5. `if` $(e = a[i])$ `then` $y := \mathbb{T}$; `break`;
6. $i := i + 1$
7. `return`$[y]$

algorithm of the element e in the array a and returns $\mathbb{T}$ (true), if the element was found, or $\mathbb{F}$ (false), otherwise. The loop is manually annotated with the invariant

$$I[i,y] :\Longleftrightarrow 0 \leq i \leq n \wedge \Big(\big(\underset{0\leq j<i}{\forall} a[j] \neq e \wedge y = \mathbb{F}\big) \vee \big(a[i] = e \wedge y = \mathbb{T}\big)\Big)$$

We develop a purely logic-based approach for program analysis, meaning that the program correctness (and implicitly the loop correctness) is expressed 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 (object theories). For instance, in Algorithm 1 the object theory is the first-order theory of integer-indexed arrays.

Further, a *metatheory* (in predicate logic with equality) is constructed for reasoning about programs. While the object theory is program specific, the metatheory is general. We describe it in the following. A program is a tuple of statements. Therefore, the metatheory contains specific functions and predicates from the set theory and tuple theory, as well as appropriate function symbols for the construction of program statements (assignment, conditionals, loops, abrupt statements: `break` and `return`). The statements contain formulae and terms from the signature of the underlying object theory. For instance, the condition of the `if` statement in Algorithm 1 is expressed in the object theory. Thus, there is no translation between the logical model of the program and the logical expressions occurring in the program.

The program statements and the program itself are meta-terms because they are composed of meta- and object level constructs. The terms and the formulae from the object theory are also *meta-terms* from the point of view of the metatheory and they are considered quoted.

Furthermore, the metatheory contains the properties of a meta-predicate for syntax checking and meta-functions for semantics and verification conditions generation, including termination condition. They are constructed as inductive definitions and use forward symbolic execution in their specific task (see [6] for a complete description).

Syntax. For loops, the syntax checker inspects, additionally to syntactic correctness, that each variable is initialized. The motivation behind this additional requirement is to avoid the misuse of variables values.

Semantics. We define the loop semantics as an implicit definition of a function, conventionally denoted by f. Our semantics is denotational, because it translates the loop into logical formulas, whose semantics is already known.

$$\underset{i,y:I[i,y]}{\forall} f[i,y] = \begin{cases} \langle i, y\rangle & \text{if } i \geq n \\ \langle i, \mathbb{T}\rangle & \text{if } i < n \wedge e = a[i] \\ f[i+1, y] & \text{if } i < n \wedge e \neq a[i] \end{cases} \quad (1)$$

Remark 1. Note that the original loop is translated into a *function*. From this point on, one could reason about the `while` loops using the Scott fixpoint theory [18, p.86], however we follow a logic-based approach.

The semantics of Algorithm 1 is (1). It was obtained by analyzing the program lines: $\langle 4\rangle$ (first branch) and $\langle 4, 5\rangle$ (second branch), respectively $\langle 4, 5, 6\rangle$ (third branch). (1) states the following: "For all values of the critical variables i and y satisfying the invariant $I[i, y]$, if the `while` loop condition $i < n$ does not hold then the value of the function f is the tuple $\langle i, y\rangle$, representing the current values of i and y; otherwise, if the element e is at the position i in the array then the value of f is the tuple $\langle i, \mathbb{T}\rangle$, otherwise, if the element e is not at the position i in the array then the value of f is computed recursively w.r.t. the next iteration."

Partial Correctness. A meta-function generates partial correctness conditions:

- safety conditions: loops are called with appropriate values of the critical variables
- functional conditions: the output condition of the program is a consequence of the accumulated conditions on the respective execution path.

For example, the fact that the invariant is inductively preserved by the execution of the loop body is a safety condition, that is:

$$0 \le i \le n \wedge \big((\underset{0 \le j < i}{\forall} a[j] \neq e \wedge y = \mathbb{F}) \vee (a[i] = e \wedge y = \mathbb{T}) \big) \wedge i < n \implies 0 \le i < n \wedge \big((\underset{0 \le j \le i}{\forall} a[j] \neq e \wedge y = \mathbb{F}) \vee (a[i+1] = e \wedge y = \mathbb{T}) \big)$$

Termination. A meta-function generates a termination condition formula for each loop, an induction principle developed from the structure of the loop.

For Algorithm 1, the termination condition is:

$$\underset{i,y:I[i,y]}{\forall} \wedge \begin{cases} i \ge n \Rightarrow \pi[i,y] \\ i < n \wedge (e = a[i]) \Rightarrow \pi[i,\mathbb{T}] \\ i < n \wedge (e \neq a[i]) \wedge \pi[i+1,y] \Rightarrow \pi[i,y] \end{cases} \Rightarrow \underset{i,y:I[i,y]}{\forall} \pi[i,y] \quad (2)$$

and was obtained by an analysis similar to the semantics function.

In (2), π is a *new constant symbol*, thus it behaves like a universally quantified predicate. The formula consists of an implication between two universally quantified parts, both over the critical variables i and y which satisfies the loop invariant $I[i,y]$. The left-hand side is a conjunction of implicational clauses. The first two clauses correspond to the branch where the loop terminates due to loop exit, respectively, the element e is found in the array at position i. On the last clause, the loop terminates because the element e is not found at position i and because the loop terminates at the next iteration. The rationale behind (2) is as follows. Let us consider the predicate $\tau[i,y]$ named "the loop terminates on the values i and y", whose logical definition is not actually known. 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:

1) the loop condition $i < n$ does not hold,
2) the element e is found in the array at position i,
3) the element e is not found in the array at position i and it terminates at the next iteration, for the value $i := i+1$.

We consider that the predicate expressing termination is the strongest predicate obeying this property T. The termination condition states that the invariant $I[i,y]$ 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 variables which fulfill the invariant, in particular, for the values of the critical variables at the entry point of the loop because they preserve the invariant.

III. Soundness of the Method: Automated Formal Proof in the *Theorema* system

The formalization, implementation and automated proofs of soundness of our method are performed in the *Theorema* system [7]. The system was built with the goal of providing one logical and software system frame for the entire process of mathematical exploration process. Implemented on top of the computer algebra system Mathematica [30], the language of *Theorema* is higher-order predicate logic extended with sequence variables. The main reason for choosing *Theorema* system in our work, instead of other proving systems, lies in its ability to present the concepts and to perform proofs of them in natural style, that is similar to the human style. The proofs were carried out by extending the capabilities of the predicate logic prover of the system.

Existing approaches prove correctness of loops in a Hoare-like logic. For example

$$\frac{\{I \wedge b\} \quad c \quad \{I\}}{\{I\} \quad \underline{\texttt{while}}\, b\, \underline{\texttt{do}}\, c \quad \{I \wedge \neg b\}},$$

has the following interpretation: "To show that, if before the execution of a while loop the property I holds and after its termination the property $I \wedge \neg b$ holds, unless it aborts or runs forever, it suffices to show that the property I is preserved by the execution of the command c, execution which is performed when also b holds."

For abruptly terminating loops, the correctness notion has to be extended to include the notion of abnormal state: "If the execution of c starts in a state satisfying I, then the execution of c terminates abruptly in a state satisfying $I \wedge \neg b$". To this end, the notion of abnormal correctness is introduced (see [10] for an approach).

Unlike these approaches, we prove the correctness of the loops in the object logic of the program, hence no mixture between programming constructs (meta-level) and logical formulas (object level) exists. The separation between meta- and object level makes our proofs simpler and easier to automate.

A. Simple Loops

In this section we prove the correctness of loops without abrupt termination. Such a loop can be brought into the following form:

$$\underline{\texttt{while}} \quad \phi[\delta] \quad \underline{\texttt{do}} \quad \delta{:=}R[\delta], \quad (3)$$

annotated with the loop invariant $\iota[\delta]$, where $\phi[\delta]$, and $R[\delta]$ are the loop condition and the function representing the update of the critical variable δ performed in the loop body, respectively. For example, in Algorithm 1, $\phi[i]$ is $i < n$ and $R[i]$ is $i+1$.

While loop (3) has the semantics (4), partial correctness (safety) (5) and termination condition (6).

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] = \begin{cases} \delta & \text{if} \quad \neg\phi[\delta] \\ f[R[\delta]] & \text{if} \quad \phi[\delta] \end{cases} \quad (4)$$

$$\underset{\delta:\iota[\delta]}{\forall} \iota[R[\delta]] \quad (5)$$

$$\underset{\delta:\iota[\delta]}{\forall} \wedge \left\{ \begin{array}{l} \neg\phi[\delta] \Rightarrow \pi[\delta] \\ \phi[\delta] \wedge \pi[R[\delta]] \Rightarrow \pi[\delta] \end{array} \right\} \Rightarrow \underset{\delta:\iota[\delta]}{\forall} \pi[\delta] \quad (6)$$

The total correctness of simple while loops "The loop invariant is always preserved." is expressed formally as:

$$\underset{\delta:\iota[\delta]}{\forall} \iota[f[\delta]]. \tag{7}$$

We express the soundness of the verification method for loops of type (3) as follows: "Formula (7) is a logical consequence of the semantics (4) and of the termination condition (6)."

Note that a function like in (4) always exists but does not necessary terminate. However, we still prove explicitly its existence (and uniqueness) based on a witness term. The fact that the witness has a closed-form solution is important for the simplicity of the proofs. Therefore, we prove first that the existence and uniqueness of an f satisfying (4) is a logical consequence of the verification conditions. The semantics formula (4) is expressed in terms of the repetition function R and of the recursion index, a natural number expressing how many times the loop is iterated. Therefore, the existence of the function R and of the recursion index has to be proved beforehand.

Summarizing, in order to prove the correctness of loops (Theorem 1), one needs to prove:

- existence of the repetition function (Lemma 1),
- existence of the recursion index (Lemma 2),
- existence and uniqueness of the function implemented by the loop (Lemma 3).

(Notation: $\mathbb{N}$ denotes the natural numbers, and $^+$, $^-$ denote the successor, respectively the predecessor functions.)

Lemma 1. *(Existence of the repetition function) Formula*

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

is a logical consequence of the natural number theory.

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

Remark 2. We use $h^n[x]$, instead of $G[n,x]$, in our formalism.

Remark 3. It is straightforward to show that $h^n[h[x]] = h^{n^+}[x]$.

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

Lemma 2. *(Existence of the recursion index) Formula*

$$\underset{\delta:\iota[\delta]}{\forall}\ \underset{n}{\exists}\Big(\neg\phi[R^n[\delta]] \wedge \underset{m}{\forall}\ \big(\neg\phi[R^m[\delta]] \Rightarrow m \geq n\big)\Big)$$

is a logical consequence of the termination condition (6).

Proof sketch. The automated proof uses a built-in natural induction principle. Additionally, the following assumptions are used:

$$\underset{x}{\forall}\ R^0[x] := x \tag{8a}$$

$$\underset{x,n}{\forall}\ R^n[R[x]] := R^{n^+}[x] \tag{8b}$$

$$\underset{n}{\forall}\ n \geq 0 \tag{8c}$$

$$\underset{n\neq 0}{\forall}\ (n^-)^+ := n \tag{8d}$$

$$\underset{m,n}{\forall}\ m \geq n \Rightarrow m^+ \geq n^+ \tag{8e}$$

Remark 4. From Lemma 2, one can see immediately that n is unique, thus, by Skolemization, one obtains the function $M[\delta]$ called the recursion index of δ, that is:
$M[\delta] := \{n \mid (\neg\phi[R^n[\delta]] \wedge \underset{m:\mathbb{N}}{\forall}\ (\neg\phi[R^m[\delta]] \Rightarrow m \geq n)\}$.

Lemma 3. *(Existence and uniqueness of the function implemented by the loop) The existence and uniqueness of an f satisfying formula* (4) *is a logical consequence of the termination condition* (6) *and of the safety verification condition* (5).

Proof sketch. For proving the existence, one takes $\underset{\delta:\iota[\delta]}{\forall} f[\delta] := R^{M[\delta]}[\delta]$ as witness for the loop semantics and derives the expression of f on each execution branch as required by (4) The proof requires also the use of (8a), (8b) and:

$$\underset{\delta:\iota[\delta]}{\forall}\ (\neg\phi[\delta] \Rightarrow M[\delta] := 0) \tag{9a}$$

$$\underset{\delta:\iota[\delta]}{\forall}\ (M[R[\delta]]^+ := M[\delta]) \tag{9b}$$

For proving the uniqueness, one takes two different semantics functions, e.g. f and g, of the form (4) and shows that they are the same.

Remark 5. Note that a total function f as in (4) always exists, but it is not necessarily unique. Its uniqueness comes from the termination condition.

Theorem 1. *(Correctness of simple loops) Formula* (7) *is a logical consequence of the semantics formula* (4) *and of the termination condition* (6).

Proof sketch. The proof is straightforward by taking in (6) $\pi[\delta]$ as $\iota[f[\delta]]$. This is because the left-hand side of the (6) becomes identical to the functional conditions generated for partial correctness.

Remark 6. Theorem (1) can be proved also by using the semantics witness from Theorem 3. In the respective proof, one needs information about the loop semantics on different execution branches as given by (4).

B. Abruptly Terminating Loops

There are basically two methods of proving the correctness of an abruptly terminating loop:

1) prove its correctness directly;
2) transform it into an equivalent simple one and prove the total correctness of the transformed version.

The drawback in the first case is that the invariant might become difficult to express and too lengthy for loops with

many ramifications and abrupt statements. In the second case, the difficulty might arise at program transformation, but the gain is that the invariants are simpler and the correctness of the initial loop resumes to the correctness of a loop-free construct due to the fact that the correctness of simple loops was already proved (Section III-A).

We chose to prove correctness by the second method.

1) Non-nested Abruptly Terminating Loops. Case `break`*:* Any `while` loop abruptly terminating via `break` can be expressed as in Example 1 even if it contains other loops with `break`; `break` from a inner loop can be eliminated and the inner loop can be expressed as function call.

Example 1.

```
while φ[δ] do
  if ψ[δ] then
    δ := S[δ];
    break
  else
    δ := R[δ]
```

Example 2.

```
while φ[δ] ∧ ¬ψ[δ] do
      δ := R[δ];
if φ[δ] ∧ ψ[δ] then
      δ := S[δ]
```

For instance, Example 1 is transformed into Example 2.

Each loop is annotated with an invariant. Note that, in general, the invariants of Examples 1 and 2 are not the same, namely the invariant of Example 1 is stronger. However, we use this invariant for both loops (and we refer to it as $\iota[\delta]$) because it implies also the invariant of the loop in Example 2. The same holds for Examples 1 and 2.

Like for simple loops, the correctness of abruptly terminating `while` loops via `break` resumes to proving the soundness of the method. In this case, proving soundness reduces to show the equivalence of semantics functions of Examples 1 and 2 (Lemma 4).

Let (10) be the semantics of Example 1 and (11) a witness satisfying it.

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] = \begin{cases} \delta & \text{if } \neg\phi[\delta] \\ S[\delta] & \text{if } \phi[\delta] \wedge \psi[\delta] \\ f[R[\delta]] & \text{if } \phi[\delta] \wedge \neg\psi[\delta] \end{cases} \tag{10}$$

$$\underset{\delta:\iota[\delta]}{\forall} f[\delta] := \begin{cases} R^{M[\delta]}[\delta] & \text{if } \neg(\phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \\ S[R^{M[\delta]}[\delta]] & \text{if } \phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]] \end{cases} \tag{11}$$

where $M[\delta] := \Big\{ n | \neg\left(\phi[R^n[\delta]] \wedge \neg\psi[R^n[\delta]]\right) \wedge \Big(\underset{m:\mathbb{N}}{\forall} \neg\left(\phi[R^m[\delta]] \wedge \neg\psi[R^m[\delta]]\right) \Rightarrow m \geq n \Big) \Big\}$ is the recursion index of the loop. Further, let (12) and (13) be the semantics of the simple loop and, respectively, of the conditional obtained of Example 2.

$$\underset{\delta:\iota[\delta]}{\forall} f'[\delta] = \begin{cases} \delta & \text{if } \neg(\phi[\delta] \wedge \neg\psi[\delta]) \\ f'[R[\delta]] & \text{if } \phi[\delta] \wedge \neg\psi[\delta] \end{cases} \tag{12}$$

$$\underset{\delta:\iota[\delta]}{\forall} g'[\delta] = \begin{cases} \delta & \text{if } \neg\phi[\delta] \\ S[\delta] & \text{if } \phi[\delta] \wedge \psi[\delta] \end{cases} \tag{13}$$

Let (14) and (15) be witnesses satisfying (12), respectively, (13).

$$\underset{\delta:\iota[\delta]}{\forall} f'[\delta] := R^{M[\delta]}[\delta] \tag{14}$$

$$\underset{\delta:\iota[\delta]}{\forall} g'[\delta] := \begin{cases} \delta & \text{if } \neg(\phi[\delta] \wedge \psi[\delta]) \\ S[\delta] & \text{if } \phi[\delta] \wedge \psi[\delta] \end{cases} \tag{15}$$

The semantics witness of Example 2 is $F'[\delta] = g'[f'[\delta]]$ and is obtained by composing the semantics witnesses (14) and (15). We have

$$\underset{\delta:\iota[\delta]}{\forall} F'[\delta] := \begin{cases} R^{M[\delta]}[\delta] & \text{if } \neg(\phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \\ S[R^{M[\delta]}[\delta]] & \text{if } \phi[R^{M[\delta]}[\delta]] \wedge \psi[R^{M[\delta]}[\delta]]) \end{cases} \tag{16}$$

Lemma 4. *Examples 1 and 2 implement the same semantics function.*

Proof sketch. The proof is immediate by observing that (11) and (16) are the same.

2) Non-nested Abruptly Terminating Loops. Case `return`*:* Any `while` loop abruptly terminating via `return` can be expressed as in Example 3 even if it contains other loops with `break` and `return`; both `break` and `return` from a inner loop can be eliminated and the inner loop can be expressed as function call.

Example 3.

```
while φ[δ] do
  if ψ[δ] then
    δ := S[δ];
    return[δ]
  else
    δ := R[δ]
```

Example 4.

```
while φ[δ] ∧ ¬ψ[δ] do
      δ := R[δ];
if φ[δ] ∧ ψ[δ] then
      δ := S[δ]
      return[δ]
```

For instance, Example 3 is transformed into Example 4.

The correctness of loops abruptly terminating via `return` can be proved following the principles of loop abruptly terminating via `break`, with the remark that the `return` statement causes execution to exit the program. Hence, additionally to proving the equivalence of the semantics functions of Examples 3 and 4, one has to prove that the output condition of the program holds upon the execution of the `return`.

3) Nested Abruptly Terminating Loops: Our approach can be extended to arbitrarily nested, abruptly terminating `while` loops. The proofs are similar to those with non-nestedness, the effort is to transform the initial loops into simple loops. A naive algorithm for such a translation is:

1) analyze the program top-down detecting the innermost loop with abrupt termination,
2) transform it into a normal terminating one; the abrupt statements are eliminated in the order they appear: `break` is eliminated from the currently analyzed loop, from the all wrapper loops and from the program itself, `return` is eliminated only from the currently analyzed loop,
3) repeat 1. and 2. until there are no loops with abrupt termination.

Of course, the program text which does not need transformations is copied correspondingly.

We apply our method to Algorithm 2, search of an element into a bidimensional array.

Algorithm 2 Search in a bidimensional array

1. in a: array of integers;
   m, n, e:integers $m \geq 0$, $n \geq 0$
2. out β: integer or β_1, β_2: integers where
   $\Big(\big(\exists\limits_{0\leq k<m}\ \exists\limits_{0\leq l<n} a[k][l] = e\big) \wedge (a[\beta_1][\beta_2] = e)\Big) \vee$
   $\Big(\big(\forall\limits_{0\leq k<m}\ \forall\limits_{0\leq l<n} a_[k][l] \neq e\big) \wedge (\beta = -1)\Big)$
3. $i := 0$; $j := 0$;
4. while $(i < m)$ do
5. $\quad j := 0$;
6. $\quad$ while $(j < n)$ do
7. $\quad\quad$ if $(e = a[i][j])$ then return$[i, j]$;
8. $\quad\quad j := j + 1$;
9. $\quad i := i + 1$;
10. return[-1]

The translated version is as follows

$i := 0$; $j := 0$;
while $\Big(i < m \wedge \neg\big(j < n \wedge (e = a[i][j])\big)\Big)$ do
$\quad j := 0$;
$\quad$ while $\big(j < n \wedge (e \neq a[i][j])\big)$ do
$\quad\quad j := j + 1$;
$\quad i := i + 1$;
if $((i < m \wedge j < n \wedge (e = a[i][j]))$ then
$\quad$ return$[i, j]$;
return $[-1]$;

The abrupt termination via return was transferred to the main program. The correctness of the simple loops is proved as follows.

1) Prove the correctness of the inner loop.
2) Prove the correctness of the wrapper loop by considering the inner loop as a black-box characterized by the loop invariant the loop invariant is used in the proof of correctness of the wrapper loop.

IV. Conclusions

We showed that reasoning about imperative programs, in particular also about the ones including loops, does not necessarily need a complex theoretical construction, because it is possible to transfer the semantics of the program into the semantics of the logical formulas, thus avoiding any special theory related to program execution. Moreover, even the termination condition can be expressed as a logical formula in the object theory of the domain manipulated by the program. In our approach, this condition is in fact equivalent to an induction principle, which makes it very instrumental in proving the existence and uniqueness of the function implemented by the loop.

In computer science, when a new program is written then one needs to prove that it is correct. In mathematics, when a new function is defined (implicitly), then one needs to prove that it exists and it is unique. Our approach shows that these two processes are essentially the same: if the semantics of a program is defined in a logical functional way (as the function implemented by the program), then the total correctness of the program is equivalent to the existence and uniqueness of the function implemented by the program. This applies as well to recursive programs [6].

Acknowledgment

The first author is supported by a DOC-fFORTE-fellowship of the Austrian Academy of Sciences.

References

[1] B. Beckert, R. Hähnle, and P. Schmitt (eds.), *Verification of Object-Oriented Software: The KeY Approach*, Springer, 2007.
[2] J. Berg and B. Jacobs, *The LOOP Compiler for Java and JML*, Proceedings of TACAS, 2001.
[3] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004.
[4] R. Boute, *Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus*, ACM Transactions on Programming Languages and Systems (2006).
[5] B. Cook, A. Podelski, and A. Rybalchenko, *Termination Proofs for Systems Code*, ACM SIGPLAN Notices (2006).
[6] M. Erascu and T. Jebelean, *A Purely Logical Approach to Imperative Program Verification*, Tech. report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2010.
[7] B. Buchberger et al., *Theorema: Towards Computer-Aided Mathematical Theory Exploration*, Journal of Applied Logic (2006).
[8] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, *Extended Static Checking for Java*, ACM SIGPLAN Notices (2002).
[9] M. Gordon and T. Melham (eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
[10] M. Huisman, *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*, Ph.D. thesis, University of Nijmegen, 2001.
[11] I. T. Kassios, P. Müller, and M. Schwerhoff, *Comparing Verification Condition Generation with Symbolic Execution: An Experience Report*, Proceedings of VSTTE, 2012.
[12] M. Kaufmann, J. Strother Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.
[13] J. King, *Symbolic Execution and Program Testing*, Communications of the ACM (1976).
[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, 2002.
[16] G. Leavens and Y. Cheon, *Design by Contract with JML*, 2003.
[17] C. S. Lee, N. Jones, and A. Ben-Amram, *The Size-Change Principle for Program Termination*, ACM SIGPLAN Notices (2001).
[18] J. Loeckx, K. Sieber, and R. Stansifer, *The Foundations of Program Verification*, John Wiley & Sons, Inc., 1984.
[19] J. McCarthy, *A Basis for a Mathematical Theory of Computation*, Computer Programming and Formal Systems, 1963.
[20] J. Meyer and A. Poetzsch-Heffter, *An Architecture for Interactive Program Provers*, Proceedings of TACAS, 2000.
[21] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Springer, 2002.
[22] M. Norrish, *C Formalized in HOL*, Ph.D. thesis, Cambridge University, 1998.
[23] O. Olsson and A. Wallenburg, *Customised Induction Rules for Proving Correctness of Imperative Programs*, Proceedings of SEFM, 2005.
[24] S. Owre, J. Rushby, and N. Shankar, *PVS: A Prototype Verification System*, Proceedings of CADE, 1992.
[25] A. Poetzsch-Heffter and P. Müller, *A Programming Logic for Sequential Java*, Proceedings of ESOP, 1999.

[26] W. Schreiner, *Understanding Programs*, Tech. report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2008.
[27] K. Slind, *Function Definition in Higher-Order Logic*, Proceedings of TPHOL, 1996.
[28] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1981.
[29] D. von Oheimb, *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*, Ph.D. thesis, Technische Universität München, 2001.
[30] S. Wolfram, *The Mathematica Book. Version 5.0*, Wolfram Media, 2003.