Abstract This paper shows how classic inductive assertions can be used in conjunction with a formal operational semantics to prove partial correctness properties of programs. The method imposes only the proof obligations that would be produced by a verification condition generator – but does not require the definition of a verification condition generator. All that is required is a theorem prover, a formal operational semantics, and the object program with appropriate assertions at user-selected cut points. The verification conditions are generated in the course of the theorem-proving process by straightforward symbolic evaluation of the formal operational semantics. The technique is demonstrated by proving the partial correctness of simple bytecode programs with respect to a preexisting operational model of the Java Virtual Machine.

Keywords Software verification · Theorem proving · Verification condition · JVM

1 Summary

This paper connects two well-known approaches to program verification: operational semantics and inductive assertions. The paper shows how one can adopt the clarity and concreteness of a formal operational semantics while incurring just the proof obligations of the inductive assertion method, without writing a verification condition generator or other extralogical tool. In particular, the formal definition of the state transition function can be used directly to generate verification conditions for annotated programs.

In this section the idea is presented in the abstract. Some details are skipped, and a deliberate confusion of states with formulas is perpetrated to convey the basic idea.

<table>
<thead>
<tr>
<th>labels</th>
<th>program π</th>
<th>paths</th>
<th>assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td></td>
<td>f(s)</td>
<td>P(s)</td>
</tr>
<tr>
<td>β</td>
<td></td>
<td>t</td>
<td>R(s)</td>
</tr>
<tr>
<td>γ</td>
<td>HALT</td>
<td>g(s)</td>
<td>Q(s)</td>
</tr>
</tbody>
</table>

Subsequently, the method is applied to a particular formal operational semantics, program, annotation, mechanical theorem prover, etc., to demonstrate that the basic idea is practical.

Consider a simple one-loop program \( \pi \) (Fig. 1) that concludes with a `HALT` instruction. Assume instructions are addressed sequentially, with \( \alpha \) being the address or label of the first instruction and \( \gamma \) being the address or label of the `HALT`. Let the pre- and postconditions of the program be \( P \) and \( Q \), respectively. The arrows of Fig. 1 indicate the control flow; functions \( f, g, \) and \( h \) indicate the compound state transitions along the arcs, and \( t \) is the test for staying in the loop. \( R \) is the loop invariant and “cuts” the only loop. The partial correctness challenge is to prove that if \( P \) holds at \( \alpha \), then \( Q \) holds whenever (if) control reaches \( \gamma \).

To give meaning to such programs with an operational semantics, one formalizes the abstract machine state and the effect of each instruction on the state. Typically the state, \( s \), is a vector or \( n \)-tuple describing available computational resources such as environments, stacks, flags, etc. It is assumed here that the state includes a program counter, \( pc(s) \), and the current program, \( prog(s) \), which are used to

Fig. 1 One-loop program \( \pi \) with annotations
determine the next instruction. Instructions are given meaning by defining a state transition function step. Typically, step(s) is defined by considering the next instruction and transforming the state components accordingly. For example, a LOAD instruction might advance the program counter and push onto some stack the contents of some specified variable. More complicated instructions, such as method invocation, may affect many parts of the state. The HALT instruction is particularly simple; it is a no-op.

It is convenient to define an iterated step function:

\[
\text{run}(k, s) = \begin{cases} 
  s & \text{if } k = 0 \\
  \text{run}(k - 1, \text{step}(s)) & \text{otherwise}
\end{cases}
\]

and to make the convention that \(s_k = \text{run}(k, s)\).

Given this operational semantics, the formalization of the partial correctness result is

**Theorem** Correctness of Program 1.

\(\text{pc}(s) = \alpha \land \text{prog}(s) = \pi \land \text{pc}(s_k) = \gamma \rightarrow Q(s_k)\).

**Proof** In an operational semantics setting, theorems such as the Correctness of Program 1 are proved by establishing an invariance \(\text{Inv}(s)\) with the following three properties:

1. \(\text{Inv}(s) \rightarrow \text{Inv}(\text{step}(s))\),
2. \(\text{pc}(s) = \alpha \land \text{prog}(s) = \pi \land \text{pc}(s_k) = \gamma \rightarrow \text{Inv}(s_k)\), and
3. \(\text{pc}(s) = \gamma \land \text{prog}(s) = \pi \land \text{Inv}(s_k) \rightarrow Q(s_k)\).

The main theorem is then proved as follows. The inductive application of property 1 produces

4. \(\text{Inv}(s) \rightarrow \text{Inv}(s_k)\).

Furthermore, instantiation of the \(s\) in property 3 with \(s_k\) produces

5. \(\text{pc}(s_k) = \gamma \land \text{prog}(s_k) = \pi \land \text{Inv}(s_k) \rightarrow Q(s_k)\).

We assume no instruction in \(\pi\) changes the program; hence \(\text{prog}(s) = \text{prog}(s_k)\). The Correctness of Program 1 then follows immediately from 2, 4, and 5.

**Property 1** above is problematic; it forces the user of the methodology to characterize all the states reachable from the chosen initial state. Contrast this situation with that enjoyed by the user of the inductive assertion method, where assertions are attached only to certain user-chosen cut points, as in Fig. 1. An extralogical process, which encodes the language semantics as formula transformations, is then applied to the annotated program text to generate proof obligations or verification conditions:

- VC1. \(P(s) \rightarrow R(f(s))\),
- VC2. \(R(s) \land t \rightarrow R(g(s))\), and
- VC3. \(R(s) \land \neg t \rightarrow Q(h(s))\).

If these formulas are proved, the user is then assured that, if \(P\) holds initially, then \(Q\) holds when (if) the program terminates.

To render this assurance formal, i.e., write it as a formula, one typically adopts some logic of programs, i.e., a logic that allows the combination of classical mathematical expressions about numbers, sequences, vectors, etc., with program text and terminology. The resulting programming language semantics is extralogical in the sense that it is expressed as rules of inference in a metalanguage and is not directly subject to formal analysis within the logic.\(^1\) In contrast, in the operational approach, the semantics is expressed within the language (typically as defined functions or relations on states), programs are objects in the logical universe, and the properties of both – programs and the semantic functions and relations – are subject to proof within the logic.

The central question of this paper is whether it is possible to have the best of both worlds: the concreteness and clarity of an operational semantics in a classical logical setting but the elegance and simplicity of an inductive assertion-style proof. The central question may be put bluntly as “Is it possible to prove the formula named ‘Correctness of Program 1’ above directly from VC1–VC3?” The answer is yes.

Recall that the proof of Correctness of Program 1 required the definition of \(\text{Inv}(s)\) satisfying properties 1–3 above. The key to constructing an inductive assertion-style proof in an operational setting is the following definition of \(\text{Inv}(s)\).

\[\text{Inv}(s) \equiv \begin{cases} 
  \text{prog}(s) = \pi \land \text{pc}(s) = \alpha, \quad \text{if } \text{pc}(s) = \alpha, \\
  \text{prog}(s) = \pi \land \text{pc}(s) = \beta, \quad \text{if } \text{pc}(s) = \beta, \\
  \text{Inv}(\text{step}(s)) \quad \text{otherwise}.
\end{cases}\]

The logician will immediately ask whether there exists a predicate satisfying this equivalence. The affirmative answer is provided in [12]. The logical crux of the matter is that \(\text{Inv}(s)\) is defined with tail recursion and there exists a satisfying and total witness for every tail-recursive equivalence. If some loop in the program is not cut, the equivalence may not uniquely define a predicate, but at least one witness exists.

\(\text{Inv}(s)\) clearly has properties 2 and 3. It therefore remains only to prove property 1. As will become apparent, the proof that \(\text{Inv}(s)\) has property 1 will generate the verification conditions as subgoals. To drive this home, we describe the process by which the proof is constructed rather than merely the formulas produced. Recall Fig. 1. Successive steps from a state \(s\) with \(\text{pc}(s) = \alpha\) eventually produce the state \(f(s)\) with \(\text{pc}(\beta)\). Similarly, if \(t\), then successive steps from a state \(s\) with \(\text{pc}(s) = \beta\) produce \(g(s)\) with \(\text{pc}(\alpha)\), and if \(\neg t\), then successive steps from a state \(s\) with \(\text{pc}(s) = \beta\) produce \(h(s)\) with \(\text{pc}(\gamma)\). Furthermore, repeated symbolic expansion and simplification of the step function produce the transformations described by \(f, g,\) and \(h\).

**Theorem** Property 1.

\(\text{Inv}(s) \rightarrow \text{Inv}(\text{step}(s))\)

**Proof** Consider the cases on \(\text{pc}(s)\) as used in the definition of \(\text{Inv}\).

**Case** \(\text{pc}(s) = \alpha\). The hypothesis \(\text{Inv}(s)\) may be simplified to \(\text{prog}(s) = \pi \land \text{pc}(s)\). Consider the conclusion, \(\text{Inv}(\text{step}(s))\). Symbolic simplification of \(\text{step}(s)\), given \(\text{pc}(s) = \alpha\) and

\(^1\) See, however, the discussion of [3] in the next section.