Inductive Assertions and Operational Semantics

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
E-mail: moore@cs.utexas.edu

Abstract. This paper shows how classic inductive assertions can be used in conjunction with an 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 generation. The paper focuses on iterative programs but recursive programs are briefly discussed. Assertions are attached to the program by defining a predicate on states. This predicate is then "completed" to an alleged invariant by the definition of a partial function defined in terms of the state transition function of the operational semantics. If this alleged invariant can be proved to be an invariant under the state transition function, it follows that the assertions are true every time they are encountered in execution and thus that the post-condition is true if reached from a state satisfying the pre-condition. But because of the manner in which the alleged invariant is defined, the verification conditions are sufficient to prove invariance. Indeed, the "natural" proof generates the classical verification conditions as subgoals. The invariant function may be thought of as a state-based verification condition generator for the annotated program. The method allows standard inductive assertion style proofs to be constructed directly in an operational semantics setting. The technique is demonstrated by proving the partial correctness of simple bytecode programs with respect to a pre-existing operational model of the Java Virtual Machine.

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 extra-logical 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. 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 π (Figure 1) that concludes with a `HALT` instruction. Assume instructions are addressed sequentially, with α being the address or label of the first instruction and γ being the address or label of the `HALT`. Let the pre- and post-conditions of the program be P and Q respectively. The arrows of Figure 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 α then Q holds whenever (if) control reaches γ.

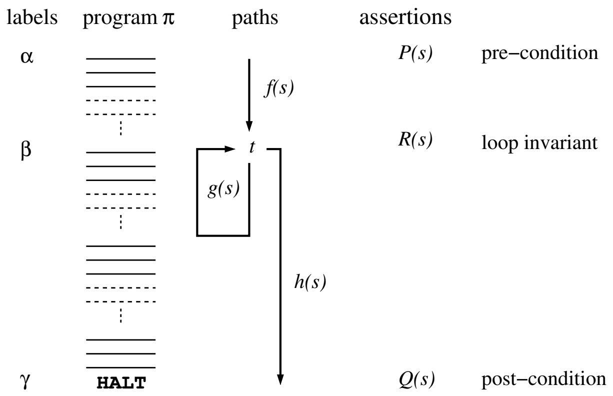

Fig. 1. The One-Loop Program π with Annotations

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 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:

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

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

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

Theorem: Correctness of Program π.

$$pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \wedge pc\,(s_k) = \gamma \rightarrow Q\,(s_k).$$

Proof. In an operational semantics setting, theorems such as the Correctness of Program π are proved by establishing an invariance $Inv\,(s)$ with the following three properties:

1. $Inv\,(s) \rightarrow Inv\,(step\,(s))$,
2. $pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \rightarrow Inv\,(s)$, and
3. $pc\,(s) = \gamma \wedge prog\,(s) = \pi \wedge Inv\,(s) \rightarrow Q\,(s)$.

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

4. $Inv\,(s) \rightarrow Inv\,(s_k)$.

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

5. $pc\,(s_k) = \gamma \wedge prog\,(s_k) = \pi \wedge Inv\,(s_k) \rightarrow Q\,(s_k)$.

We assume no instruction in π changes the program; hence $prog\,(s) = prog\,(s_k)$. The Correctness of Program π 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 Figure 1. An extra-logical 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. $R\,(s) \wedge t \rightarrow R\,(g\,(s))$, and
VC3. $R\,(s) \wedge \neg t \rightarrow Q\,(h(s))$.

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 extra-logical 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

[1] See however the discussion of [3] the next section.

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 π,' above, directly from VC1–VC3?" The answer is "yes."

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

$$\mathit{Inv}\,(s) \equiv \begin{cases} \mathit{prog}\,(s) = \pi \wedge P\,(s) & \text{if } \mathit{pc}\,(s) = \alpha \\ \mathit{prog}\,(s) = \pi \wedge R\,(s) & \text{if } \mathit{pc}\,(s) = \beta \\ \mathit{prog}\,(s) = \pi \wedge Q\,(s) & \text{if } \mathit{pc}\,(s) = \gamma \\ \mathit{Inv}\,(\mathit{step}\,(s)) & \text{otherwise} \end{cases}$$

The logician will immediately ask whether there exists a predicate satisfying this equivalence. The affirmative answer is provided in [11]. The logical crux of the matter is that $\mathit{Inv}\,(s)$ 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.

$\mathit{Inv}\,(s)$ clearly has properties 2 and 3. It therefore remains only to prove property 1. As will become apparent, the proof that $\mathit{Inv}\,(s)$ 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 Figure 1. Successive steps from a state s with pc α eventually produce the state $f\,(s)$ with pc β. Similarly, if t, then successive steps from a state s with pc β produce $g\,(s)$ with pc β, and if $\neg t$, then successive steps from a state s with pc β produce $h\,(s)$ with pc γ. Furthermore, repeated symbolic expansion and simplification of the *step* function produce the transformations described by *f*, *g*, and *h*.

Theorem: Property 1.

$$\mathit{Inv}\,(s) \rightarrow \mathit{Inv}\,(\mathit{step}\,(s))$$

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

Case: $\mathit{pc}\,(s) = \alpha$. The hypothesis, $\mathit{Inv}\,(s)$ may be simplified to $\mathit{prog}\,(s) = \pi \wedge P\,(s)$. Consider the conclusion, $\mathit{Inv}\,(\mathit{step}(s))$. Symbolic simplification of $\mathit{step}\,(s)$, given $\mathit{pc}\,(s) = \alpha$ and $\mathit{prog}\,(s) = \pi$, produces a symbolic state s' with $\mathit{pc}\,(s') = \alpha + 1$. For program π either $\alpha + 1$ is β or it is none of the cut points α, β or γ. In the latter case, $\mathit{Inv}\,(\mathit{step}\,(s)) \equiv \mathit{Inv}\,(s') \equiv \mathit{Inv}\,(\mathit{step}\,(s'))$ and stepping continues until β is reached at state $f\,(s)$. Hence, $\mathit{Inv}\,(\mathit{step}\,(s)) \equiv R\,(f\,(s'))$ (since

$prog\,(f\,(s)) = \pi$). Thus, this case simplifies to the goal

$$pc\,(s) = \alpha \wedge prog\,(s) = \pi \wedge P\,(s) \rightarrow R\,(f\,(s)).$$

This is just VC1 (with two now-irrelevant hypotheses, given traditional assertions P and R).

Case: $pc\,(s) = \beta$. The hypothesis $Inv\,(s)$ simplifies to $prog\,(s) = \pi \wedge R\,(s)$. Then the symbolic simplification of $step\,(s)$ in the conclusion produces a bifurcated symbolic state whose program counter depends on test t. Repeated expansions of the definition of *Inv* on both branches of the state eventually reach states $g\,(s)$ and $h\,(s)$ at which *Inv* is defined. The results are VC2 and VC3, respectively.

Case: $pc\,(s) = \gamma$. The hypothesis $Inv\,(s)$ simplifies to $prog\,(s) = \pi \wedge Q\,(s)$. But the $step\,(s)$ in the conclusion simplifies to s because the instruction at γ in π is the no-op `HALT`. Hence, $Inv\,(s) \equiv Inv\,(step(s))$ and this case is trivial (propositionally true independent of the assertions).

Case: otherwise. Since $pc\,(s)$ is not one of the cut-points, $Inv\,(s) \equiv Inv\,(step(s))$ by definition of *Inv* and this case is also trivial.
□

Hence, if the verification conditions VC1–VC3 have been proved, the proof of property 1, the step-wise invariance of *Inv*, involves no assertion-specific reasoning. More interestingly, given the definition of *Inv*, the proof *generates* the verification conditions by symbolic expansion of the operational semantics' state transition function.

Practically speaking this means that with a mechanical theorem prover and a formal operational semantics one can enjoy the benefits of the inductive assertion method without writing a verification condition generator or other extra-logical tools to do formula transformations.

Another practical ramification of this paper is that it provides a simple means to define a step-wise invariant given only the assertions at the cut points. Step-wise invariants are frequently needed in operational semantics-based proofs of safety and liveness properties.

2 Related Work and Discussion

McCarthy [12] made explicit the notion of operational semantics, in which "the meaning of a program is defined by its effect on the state vector."

The inductive assertion method for proving programs correct was implicitly used by von Neumann and Goldstine in [4] (1946) and Turing [18, **?**] (1949) and made explicit in the classic papers by Floyd [2] and Hoare [5]. The first mechanized verification condition generator, which generates proof obligations from code and attached assertions, was written by King [8]. Hoare, of course, rendered the inductive assertion method formal by introducing a logic of programs. From the practical perspective most program logics are mechanized with two trusted tools, a formula generator, here called a VCG ("verification condition generator"), and a theorem prover. It is not uncommon for the VCG to include not just language semantics as formula transformers but also some logical simplification (i.e., theorem proving) to keep the generated proof obligations manageable.

A notable exception is the work of Gloess [3] where the Hoare semantics of a simple imperative programming language is formalized within the higher-order logic of PVS and mechanically checked proofs of several programs are carried out with PVS. As in the present work, Gloess' proofs generate the verification conditions. The difference however is that the formal semantics is Hoare-style rather than operational and is thus designed to generate formulas.

This paper contains one apparently novel idea: a step-wise invariant can be defined from the inductive assertions using the state-transition function. One may think of this as a methodology for obtaining a state-based verification condition generator from an operational semantics. By doing it on a per program basis the method avoids the need to generate or trust extra-logical tools.

The use of inductive assertions in conjunction with a formal operational semantics to prove partial correctness results mechanically is not new. Robert S. Boyer and the author developed it for their *Analysis of Programs* course at the University of Texas at Austin as early as 1983. In that class, an operational semantics for a simple procedural language was defined in Nqthm [1] and the course explored program correctness proofs that combined operational semantics with inductive assertions. These proofs motivated the exploration of total versus partial correctness, Hoare logics, and verification condition generation. For an Nqthm proof script illustrating the use of inductive assertions in an operational semantics setting, see [13].

A recent example of the use of assertions to prove theorems about a program modeled operationally may be found in [16], where a safety property of a non-terminating multi-threaded Java system is proved with respect to an operational semantics for the Java Virtual Machine [15].

However, in the earlier work the invariant explicitly included an assertion for every value of the *pc*. (The invariant must recognize every reachable state and so must handle every *pc*; the issue is whether it does so explicitly or implicitly.)

An alternative way to combine inductive assertions at selected cut points with an operational semantics in a classical formal setting is to formalize and verify a VCG with respect to the operational semantics. In [6], for example, an HOL proof of the correctness of a VCG for a simple procedural language is described. The work includes support for mutually recursive procedures. Formal proofs of the verification conditions could, in principle, be used with the theorem stating the correctness of the VCG, to derive a property stated operationally. But the method described here does not require the definition of a VCG much less a proof of its correctness.

Logically speaking, a crucial aspect of the novel idea here is that the step-wise invariant is defined using tail recursion. The admission of a new function or predicate symbol via recursive definition is generally handled by a definitional principle that insures the existence (and often the uniqueness) of the defined concept. In many logics, this requires a termination proof. Admitting *Inv* under such a definitional principle would require a measure of the distance to the next cut point and a proof that the distance decreases under *step*. That imposes a proof burden not generally incurred by the user of the inductive assertion method. (Every loop must be cut for the inductive assertion method to be effective; the question is whether that must be proved formally or merely demonstrated by the successful generation of the verification conditions.)

The technique used here exploits the observation that *Inv* is tail-recursive and hence admissible without proof obligation, given the work of Manolios and Moore [11] in which it was proved that every tail-recursive equation may be witnessed by a total function. The tail-recursive function may not be uniquely defined by the equation — this might occur if insufficient cut points are chosen. Such a failure is manifested by an infinite loop in the process of generating/proving the step invariance. This is the same behavior a VCG user would experience in the analogous situation.

One may think of this technique as constructing a step-wise invariant from from annotations at selected cut-points. Floyd [2] described it as follows:

> It is, therefore, possible to extend a partially specified interpretation to a complete interpretation, without loss of verifiability, provided that initially there is no closed loop in the flowchart all of whose edges are not tagged and that there is no entrance which is not tagged. This offers the possibility of automatic verification of programs, the programmer merely tagging entrances and one edge in each innermost loop; the verifying program would extend the interpretation and verify it, if possible, by mechanical theorem-proving techniques.

The novelty of the idea here is merely that we "extend the interpretation" by definition of a function in the logic rather than by extra-logical means.

The technique here is similar in spirit to one used by Pete Manolios [private communication] to attack the 2-Job version of the Apprentice problem [16]. There, he defined the reachable states of the Apprentice problem as all the states that could be reached from certain states by the execution of a fixed maximum number of steps.

3 A Demonstration of the Method

To illustrate the technique a mechanized formal logic and an operational semantics must be introduced. In this paper we use the ACL2 logic [7]. In this logic, function application is denoted as in Lisp, e.g., $run\,(k, s)$ is written `(run k s)`. We will explain all the formulas as necessary.

For the demonstration we choose a pre-existing operational semantics for a significant fragment of the JVM [9]. The model is written in the ACL2 logic and may be considered a Lisp implementation of a significant fragment of the JVM interpreter. The model is called M5 [15] and it was chosen simply because it was available, stable, realistic, and written in a logic for which we have a powerful automatic/interactive mechanical theorem prover. We have a much more sophisticated JVM model, called M6 [10] and we have done some of these examples in M6 as well, but the M6 model is not yet stable.

The M5 model is fairly complex, requiring about 250 ACL2 definitions consuming about 3000 lines of formalism on top of ACL2's extensive support for discrete mathematics. In addition to many other JVM data types, M5 supports Java's 32-bit twos complement integer arithmetic, here called "`int` arithmetic," in which overflow is not signaled; adding one to the most positive `int` produces the most negative `int`. An important mathematical concept (given the examples we have chosen to exhibit here) is the mathematical function `int-fix`, which takes an arbitrary integer and returns the integer represented by its low order 32-bits, interpreted in twos complement notation. That is, for integer values of n

$$\text{int-fix}(n) = \begin{cases} n \mod 2^{32} & \text{if } n \mod 2^{32} < 2^{31} \\ n \mod 2^{32} - 2^{32}, & \text{otherwise} \end{cases}$$

Since we use ACL2 notation, we write int-fix(n) as `(int-fix n)` in the formulas in this paper.

Our M5 model includes semantics for 138 bytecode instructions including those for the creation and initialization of instance objects in the heap, manipulation of static and instance fields, the invocation of static, special, and virtual methods, Java's inheritance rules for method resolution, the creation of multiple threads, and synchronization via monitors. The model is operational in the sense that it can be executed on the output of Sun's `javac` compiler (for methods in the 138-instruction subset and after transformation of the class files into ACL2 constants).

The M5 model of the JVM is a good example of an abstract machine that is sufficiently complicated that writing a VCG for it a serious and error-prone undertaking.

M5 is formalized by defining *step* and *run* functions as above. The state includes a thread table containing stacks of method invocation frames, a heap, and a class table of loaded classes. Each frame contains a pc, bytecoded program, local variables, and operand stack. The M5 `step` function takes two arguments instead of just one: `(step th s)` is the state obtained by stepping thread `th` in state `s`. The `run` function, instead of taking the number of steps, takes a list of thread identifiers, called a schedule, and steps those threads sequentially.

Symbolic simplification of this semantics is central to the idea proposed here. Consider the following bytecode sequence (in the M5 parsed byte-stream format): `(ILOAD_1) (ICONST_1)(IADD)(ISTORE_1)`. This sequence pushes the value of local variable `1` on the operand stack, pushes the constant `1`, pops the first two items off the stack and pushes their `int` sum, and pops the stack into local variable `1`. That is, the sequence corresponds to the Java assignment `a = a+1;` if `a` is an `int` allocated in local variable 1. Suppose M5 state `s` contains a thread, `th`, the active frame of thread `th` has `pc 6` and that the bytecode sequence above is positioned starting at byte offset `6` in the current program. Suppose the locals of the frame are denoted by *locals* and the operand stack by *stack*. The symbolic simplification of `(step th s)` produces a symbolic state expression in which the active frame of thread `th` has `pc 7` and operand stack `(push (nth 1` *locals*`)` *stack*`)`. If three more such steps are taken the result is a symbolic state expression in which the active frame of thread `th` has `pc 10` and the following expression, *locals'*, for its locals `(update-nth 1 (int-fix (+ (nth 1` *locals*`) 1))` *locals*`)`. Note that the symbolic expression for local `1` in this environment, `(nth 1` *locals'*`)` simplifies to `(int-fix (+ (nth 1` *locals*`) 1))` using rewrite rules about `nth` and `update-nth`.

4 An Iterative Program

Below is an M5 program written as a Lisp constant named `*program-pi*`.

```
(defconst *program-pi*
  '((ICONST_0)       ; 0
    (ISTORE_1)       ; 1                 a := 0
    (ILOAD_0)        ; 2  top of loop:
    (IFEQ 14)        ; 3                 if n=0, goto 17
    (ILOAD_1)        ; 6
    (ICONST_1)       ; 7
    (IADD)           ; 8
    (ISTORE_1)       ; 9                 a := a+1
    (ILOAD_0)        ;10
    (ICONST_2)       ;11
    (ISUB)           ;12
    (ISTORE_0)       ;13                 n := n-2
    (GOTO -12)       ;14                 goto top of loop
    (HALT)))         ;17                 halt
```

The program is a list of instructions. Each instruction is a list and corresponds to a JVM bytecode instruction. The column of numbers preceded by semi-colons is the byte offset of the instruction, starting from the top. Thus, the first occurrence of `(ILOAD_1)`, which is the fifth instruction in the program, is located at byte offset 6 because the `(IFEQ 14)` instruction before it occupies bytes three, four, and five of the program. The text further to the right is pseudo-code for the program.

Here is a quick guide to the relevant JVM instructions.

`ICONST_0`	push 0 on the operand stack
`ICONST_1`	push 1 on the operand stack
`ICONST_2`	push 2 on the operand stack
`ISTORE_0`	pop the stack into local variable 0
`ISTORE_1`	pop the stack into local variable 1
`ILOAD_0`	push local variable 0 on the operand stack
`ILOAD_1`	push local variable 1 on the operand stack
`IFEQ` n	pop the stack to obtain v and if v is 0, add n to the pc
`IADD`	pop two items and push their `int` sum
`ISUB`	pop two items and push their `int` difference
`GOTO` n	add n to the pc

The program decrements its first local, informally called `n`, by 2 and iterates until the result is 0. On each iteration it adds 1 to its second local variable, here called `a`, which is initialized to 0. The program in `*program-pi*` is essentially that generated by the Sun Java compiler on the code fragment

```
int a = 0;
while (n!=0){a=a+1;n=n-2;}
```

except for the way that the compiler codes the "top of the loop" at the bottom to save an instruction. (That trivial optimization offers no obstacle to our method, but we code the loop more naively in this example just to make the example easier to follow.)

Thus, the method computes `n`/2, henceforth written `(/ n 2)`, when `n` is even. The program does not terminate when `n` is odd.

The program is slightly simpler to deal with if it is assumed that `n` is a non-negative `int`. The program actually terminates for negative even `int`s, because Java's `int` arithmetic wraps around: the most negative `int`, `-2147483648`, is even and when it is decremented by 2 it becomes the most positive even, `2147483646`. But to avoid excessive discussion of `int` arithmetic, we limit our attention to non-negative `int`s.

For simplicity, our program concludes with the fictitious `HALT` instruction, which stops the machine. More generally, methods end with some kind of return instruction that causes execution of the caller's code to resume. We discuss procedure (method) invocation briefly later.

Let the initial value of `n` be `n0`. The goal is to prove that if `n0` is a non-negative `int` and control reaches `pc` 17, then `n0` is even and local variable `a` has the value `(/ n0 2)`. That is, if the program halts the initial input must have been even and the final answer is half that input.

5 The Assertions at the Three Cut Points

The cut points, to which assertions will be attached, are at program counters 0 (α), 2 (β), and 17 (γ). The assertions attached to these cut points were called P, R, and Q in the earlier treatment. We will define each assertion as a function. In the definitions below, think of `n0` as the initial value of `n`, and think of `n` and a as the current values of locals n and a whenever the corresponding assertion is reached.

```
(defun P (n0 n)                ; Pre-Condition
  (and (equal n n0)
       (intp n0)
       (<= 0 n0)))

(defun R (n0 n a)              ; Loop Invariant
  (and (intp n0)
       (<= 0 n0)
       (intp n)
       (if (and (<= 0 n)
                (intp a)
                (evenp n))
           (equal (int-fix (+ a (/ n 2)))
                  (/ n0 2))
         (not (evenp n)))))

(defun Q (n0 a)                ; Post-Condition
  (and (evenp n0)
       (equal a (/ n0 2))))
```

We paraphrase these three assertions below. But the reader is reminded that their content is dictated by the semantics of the JVM programming language and what we wish to prove about `*program-pi*` by the inductive assertion method. In particular, these same (or equivalent) assertions would be needed to carry out an inductive assertion proof of this bytecode program no matter how the inductive assertion method were implemented.

The Pre-Condition, `P`, asserts that the value of local variable `n` is `n0` and that `n0` is a non-negative `int`.

The Loop Invariant, `R`, asserts that `n0` is (still) a non-negative `int`, that `n` is an `int`, and that if `n` is non-negative, `a` is an `int`, and `n` is even, then the (`int` part of the) sum of `a` and half of `n` is half of `n0`; otherwise, `n` is not even.

The Post-Condition asserts that `n0` is even and `a` is half of `n0`.

By proving this particular Post-Condition we establish not only that the final value of `a` is half the initial value of `n`, but that the initial `n` must have been even. We could weaken the post-condition to omit mention of the parity of `n` and would not have had to use such a strong loop invariant.

The details of the assertions are not germane to this paper. The assertions are typical inductive assertions for such a program. These assertions were chosen to illustrate that, within the framework of such an operational semantics, we can use inductive assertions to address partial correctness of non-terminating programs including the characterization of when termination occurs.

6 Verification Conditions

Given `*program-pi*`, the informal attachment of the three assertions to the chosen cut points, and a VCG for the JVM, the following verification conditions would be produced.

```
(defthm VC1                                       ; entry to loop
  (implies (P n0 n)
           (R n0 n 0)))
(defthm VC2                                       ; loop to loop
  (implies (and (R n0 n a)
                (not (equal n 0)))
           (R n0
              (int-fix (- n 2))
              (int-fix (+ 1 a)))))
(defthm VC3                                       ; loop to exit
  (implies (and (R n0 n a)
                (equal n 0))
           (Q n0 a)))
```

These are easily proved. The challenge is: how can these three theorems be used to verify a partial correctness result for `*program-pi*` with respect to our operational semantics?

7 Attaching the Assertions to the Code

Our M5 model supports multi-threading. Our `*program-pi*` will be running in some thread, identified by the thread identifier `th` in state `s`. (In fact, in this treatment, `th` will be the only thread scheduled so the multi-threading features of the model are largely irrelevant here.)

To relate the assertions, above, to the local variables in `*program-pi*` running in thread `th` of `s` it is convenient to define

```
(defun n (th s) (loc 0 th s))
(defun a (th s) (loc 1 th s))
```

Thus, `(n th s)` is the current value of the variable `n`, i.e., the item in slot `0` of the vector of locals in the active method of thread `th`. `(Loc i th s)` is equal to `(nth i (locals (top (call-stack th s))))`, a fact which we note only to expose a tiny bit of the underlying operational model.

In the earlier treatment of the method, we defined the invariant by conjoining each assertion with the statement "$prog\,(s) = \pi$." Here we introduce an intermediate function to do this and also to retrieve the values of the two locals from the JVM state and call them `n` and `a` for clarity.

```
(defun assertion (n0 th s)
  (let ((n (n th s))
        (a (a th s)))
```

```
    (and (equal (program (top (call-stack th s))) *program-pi*)
         (case (pc (top (call-stack th s)))
           (0  (P n0 n))
           (2  (R n0 n a))
           (17 (Q n0 a))
           (otherwise nil)))))
```

The function takes three arguments: `n0`, the initial value of `n`; the thread identifier, `th`, of the thread running our program; and `s`, the current JVM state.

The `let` binds two mathematical variables, `n` and `a`, to the current values of the two corresponding local variables of the program. The `assertion` function then asserts two things. First, the program component of the topmost frame of thread `th` is our `*program-pi*`. Second, `P`, `R`, or `Q` is asserted of the appropriate values, depending on whether the program counter (`pc`) is `0`, `2` or `17`. `Nil` (false) is asserted otherwise, but this is irrelevant because we will only use `assertion` when the `pc` is one of the three cut points.

8 The Nugget: Defining the Invariant

The nugget in this paper is how to "complete" the assertions at the cut points to define a step-wise invariant.

The invariant is introduced with the `defpun` ("define partial function") utility of ACL2 described in [11]. The assertions are tested at the three cut points and all other statements inherit the invariant of the next statement. This definition is analogous to that for *Inv* in the abstract treatment, except that the invariant also takes the initial input, `n0`, and the identifier of the relevant thread, `th`.

```
(defpun Inv (n0 th s)
  (if (member (pc (top (call-stack th s)))
              '(0 2 17))
      (assertion n0 th s)
    (Inv n0 th (step th s))))
```

9 Proofs

Recall our general description of the methodology and "property 1" of *Inv*:

1. $Inv\,(s) \rightarrow Inv\,(step\,(s))$,

Here is the ACL2 expression of that theorem.

```
(defthm Property-1-of-Inv
  (implies (Inv n0 th s)
           (Inv n0 th (step th s))))
```

As noted earlier, the proof attempt generates the verification conditions, by expanding `step` and `Inv`, driving the operational model forward from cut point to cut point. [2]

If the verification conditions, `VC1`–`VC3`, have already been proved and are known to the proof engine, then the proof of `Property-1-of-Inv` is straightforward.

If the verification conditions are not known to the proof engine, the same basic symbolic manipulation techniques that generate them continue to operate to try to prove them.

Central to the process is the symbolic simplification of state expressions under the state transition function `step`.

Having proved the invariance of `Inv` under `step` the next theorem in the mechanized "methodology" corresponds to "property 4" of the earlier proof of the Correctness of Program π.

4. $Inv\,(s) \rightarrow Inv\,(s_k)$.

Here is the ACL2 expression of that result. The theorem states that `Inv` is invariant under arbitrarily long runs of the thread in question.

```
(defthm Property-4-of-Inv
  (implies (and (mono-threadedp th sched)
                (Inv n0 th s))
           (Inv n0 th (run sched s))))
```

Note that the ACL2 formula explicitly restricts attention to schedules in which thread `th` is the only running thread. Because a schedule in the M5 model is just a list of the thread identifiers to be stepped successively, this hypothesis is formalized with the following ACL2 concept.

```
(defun mono-threadedp (th sched)
  (if (endp sched)
      t
    (and (equal th (car sched))
         (mono-threadedp th (cdr sched))))).
```

The `defun` above may be read as follows. A schedule is *mono-threaded* (with respect to thread identifier *th*) if the schedule is empty or the first thread stepped is `th` and the rest of the schedule is mono-threaded.

[2] Actually, it generates slightly "cluttered" looking formulas that are equivalent to the verification conditions. The clutter has two sources. First, there are irrelevant hypotheses about the starting location of the `pc` (at 0, 2 or 18) and the identity of the program being executed (`*program-pi*`). These hypotheses were relevant during the unwinding of the operational semantics but are not relevant once the operational semantics has been driven fully from cut to cut. Second, the generated verification conditions do not use the "nice" names `n` and `a`, but rather refer to them by their operational semantic descriptions, e.g., items `0` and `1` of the locals of the topmost frame of the arbitrary state `s`.

Why must we restrict our attention to mono-threaded schedules? The reason is that property 1 of `Inv`, as formalized, is

```
(defthm Property-1-of-Inv
  (implies (Inv n0 th s)
           (Inv n0 th (step th s))))
```

which says "if `Inv` holds of thread `th` then `Inv` holds of thread `th` after stepping thread `th`." We would not need the mono-threaded restriction if we generalized this to

```
(defthm Property-1-of-Inv-generalized
  (implies (Inv n0 th s)
           (Inv n0 th (step x s)))).
```

Of course, this generalized property 1 is harder to prove (and probably requires considerable strengthening of `Inv`) since the thread `x` being stepped might interact with thread `th`. No claim is made as to the practicality of this methodology for multi-threaded programs! An obvious question is whether this technique could be used to simplify the hideous invariant used in our own proof of the Apprentice problem [16]. We simply have not had the opportunity to try that yet but we believe it will be applicable.

The point we wish to make in this paper is that the multi-threaded operational semantics can be used to derive verification conditions for single-threaded or multi-threaded applications, merely by the choice of formula we try to prove.

Returning to the mono-threaded case, from the theorem

```
(defthm Property-4-of-Inv
  (implies (and (mono-threadedp th sched)
                (Inv n0 th s))
           (Inv n0 th (run sched s))))
```

we can easily prove the following more explicit description of a partial correctness property of `*program-pi*`: Suppose the initial state, `s0`, has `pc 0`, program `*program-pi*` and satisfies the pre-condition `P`. Let `s1` be some arbitrary mono-threaded run from `s0` and suppose the `pc` in `s1` is `17`. Then `s1` satisfies the post-condition `Q`. Formally this can be written as follows.

```
(defthm Corollary-1
  (implies (and (equal (pc (top (call-stack th s0))) 0)
                (equal (program (top (call-stack th s0)))
                       *program-pi*)
                (P n0 (n th s0))
                (equal s1 (run sched s0))
                (mono-threadedp th sched)
                (equal (pc (top (call-stack th s1))) 17))
           (Q n0 (a th s1))))
```

By expanding the pre- and post-conditions we can restate the theorem to say that if `n` is a non-negative `int` and `*program-pi*` is run (in a mono-threaded fashion) on n starting at the entry to the program, and the run ever reaches the HALT statement, then n was initially even and the final value of a is half the initial value of n.

It takes ACL2 approximately 2.36 seconds (on an Intel Xeon^{TM} 2.40GHz processor) to prove all of the theorems discussed in connection with *program-pi*. The only lemmas developed for this exercise were mathematical lemmas on the properties of evenp int arithmetic when subtracting 2.

Notice what has been accomplished. Corollary-1 is a partial correctness theorem about a JVM program, formalized with an operational semantics. The creative part of the proof consisted of the definition of the three assertions. Users familiar with inductive assertions would find these assertions straightforward (requiring only a few minutes to write down). The proof of the key lemma, Property-1-of-Inv, generated (and requires the proof of) the classic verification conditions just as though a VCG for the JVM were available. But no VCG was defined. The proof does not establish termination of the code under the pre-conditions but does characterize necessary conditions to reach the HALT statement (since we specified the post-condition as we did). Finally, neither the theorem nor the proof involved counting instructions or defining what the Boyer-Moore community calls a "clock function" for the program.

10 Packaging It Up

This methodology is so routine that it can be easily packaged as a "macro" in ACL2. A macro is an input expression that expands to some other expression (or sequence of expressions) before being evaluated. We have defined the macro defspec to "package" this way of doing inductive-assertion-style proofs about our JVM model. The expression

```
(defspec pi *program-pi*  (n0) 0 17
     (( 0 (P n0 (n th s)))
      ( 2 (R n0 (n th s) (a th s)))
      (17 (Q n0 (a th s)))))
```

essentially annotates *program-pi* so that the initial value of its first (0^{th}) local is named n0, its entry pc is declared to be 0, its exit pc is declared to be 17, and the cut points and assertions are as written. Defspec expands to a sequence of defuns and defthms including the corresponding versions of the functions assertion and Inv. Then it proves the various theorems concluding with Corollary-1. The symbol pi in the defspec form above is used to generate unique names for the various functions and lemmas associated with the problem.

It is not necessary to define the cut point assertions as functions P, R, and Q. We can write instead

```
(defspec pi *program-pi*  (n0) 0 17
     ((0 (and (equal (n th s) n0)
              (intp n0)
              (<= 0 n0)))
```

```
    (2 (and (intp n0)
            (<= 0 n0)
            (intp (n th s))
            (if (and (<= 0 (n th s))
                     (intp (a th s))
                     (evenp (n th s)))
                (equal (int-fix (+ (a th s) (/ (n th s) 2)))
                       (/ n0 2))
              (not (evenp (n th s))))))
    (17 (and (evenp n0)
             (equal (a th s) (/ n0 2))))))
```

to accomplish the same ends.

11 Another Example

Here is an entertaining example of the application of the inductive assertion method. Consider the following Java program:

```
  public static int tfact(int n){
        int i = 1;
        int b = 1;
        while (i<=n){
            int j = 1;
            int a = b;
            while (j < i) {
                b = a+b;
                j++;
            };
            i++;
        };
        return b;
   }
```

This is an algorithm dealt with by Alan Turing in his 1949 paper "Checking a Large Routine," [18]. He proves the algorithm computes factorial using the inductive assertion method except he cuts every transition in the flowchart with an assertion. The routine is a simple nested loop using repeated addition to do the multiplication.

The M5 expression of the bytecode generated by Sun for this routine is

```
(defconst *Turing-Fact*
  '(
    (iconst_1)                  ;   0
    (istore_1)                  ;   1
    (iconst_1)                  ;   2
    (istore_2)                  ;   3
```

```
    (goto 27)                    ;   4 go to 31
    (iconst_1)                   ;   7 (continuation of outer loop)
    (istore_3)                   ;   8
    (iload_2)                    ;   9
    (istore 4)                   ;  10
    (goto 11)                    ;  12 go to 23
    (iload 4)                    ;  15 (continuation of inner loop)
    (iload_2)                    ;  17
    (iadd)                       ;  18
    (istore_2)                   ;  19
    (iinc 3 1)                   ;  20
    (iload_3)                    ;  23 (top of inner loop)
    (iload_1)                    ;  24
    (if_icmplt -10)              ;  25 go to 15
    (iinc 1 1)                   ;  28
    (iload_1)                    ;  31 (top of outer loop)
    (iload_0)                    ;  32
    (if_icmple -26)              ;  33 go to 7
    (iload_2)                    ;  36
    (HALT)                       ;  37
    ))
```

We have modified the code generated for `tfact` only by inserting a `HALT` in place of the compiler's `IRETURN` at location `37`.

We prove that under certain conditions the `*Turing-Fact*` program computes `(int-fix (! n))`, i.e., the low-order 32-bits of the mathematical factorial as defined by

```
(defun ! (n)             ; n! =
  (if (zp n)             ; if n=0
      1                  ;    then 1
    (* n (! (- n 1))))) ;    else n*(n-1)!
```

To prove it correct we invoke:

```
(defspec Turing-Fact *Turing-Fact* (n0) 0 37
  ((0 (and (equal n0 (loc 0 th s))
           (intp n0)
           (<= 0 n0)
           (<= 0 (int-fix (+ 1 n0)))))
   (37 (equal (top (stack (top-frame th s)))
              (int-fix (! n0))))
   (23 (TuringFact-InnerInv
        n0           ; n0
        (loc 0 th s) ; n
        (loc 1 th s) ; i
        (loc 3 th s) ; j
```

```
    (loc 4 th s) ; a
    (loc 2 th s) ; b
    ))))
```

Note that it is necessary to cut only the inner loop, as Floyd observed [2]. We do not show the assertion we used, but it was derived from Turing's annotations [18].

Turing's program was, of course, for a machine with a different word size and he explicitly ignored finite arithmetic after briefly discussing the problem. Curiously perhaps, the program does not actually work for all `int` input. Note the last conjunct in the pre-condition at 0. It is necessary that `n0` be strictly below $2^{31} - 1$. Otherwise, on what should be the last iteration as `i` reaches `n` from below, `i++` produces a negative `i` instead of an `i` exceeding `n`.

12 Method Invocation and Return

The `HALT` instruction in the previous programs is fictitious but handy. Stepping the machine while on a `HALT` leaves the machine at the `HALT`. Thus, the invariance of the exit assertion is easy to prove once the exit is reached. In realistic code, the machine does not halt but returns control to the caller and non-trivial stepping continues. A useful inductive assertion methodology must deal with call and return. This paper does not discuss call and return in detail.

On the JVM, method invocation pushes a new stack frame on the invocation stack of the active thread. Abstractly, that frame may be thought of as containing the bytecode for the newly invoked method with initial `pc 0`. The new frame contains an initially empty "operand stack" for intermediate results. When certain return instructions are executed, the topmost item, v, on the operand stack is removed, the invocation stack is popped, and v is pushed onto the operand stack of the caller.[3]

To deal with call and return via inductive assertions, two changes are made to the methodology described above. First, instead of using `run` to run the state a certain number of steps, the new function `run-to-return` is introduced, which runs a certain number of steps or until the state returns from the call depth, `d0`, at which the run was started. Second, the assertion function is changed so that the post-condition is asserted if the call depth is less than `d0`.

To deal with recursive methods, one must characterize the stack of frames created by previous recursive calls so that `return`s produce states in which continued symbolic evaluation is possible. The reason this is necessary is that our M5 model is a "little step" semantics. That is, the instruction after a procedure invocation is *not* the instruction following it in the byte stream but the first instruction in the invoked method.

This is illustrated in [14] where a recursive Java factorial method is verified. Consider the following code fragment.

```
(iload_0)
```

[3] Some forms of return implement `void` methods and return no v to the caller.

```
(iload_0)
(iconst_1)
(isub)
(invokestatic "Demo" "rfact" 1)
(imul)
(ireturn)
```

This corresponds to the Java fragment "`return n*rfact(n-1);`." When the `invokestatic` is executed, $n-1$ is on top of the stack and n is below it. Conceptually, the `invokestatic` pops $n-1$ off the stack and leaves $(n-1)!$ in its place. Then the `imul` multiplies that by n and the method returns that number. Now suppose we have traced a path down to the `invokestatic` starting at some cut point above it. What is the next instruction? Put another way, what does the JVM actually do? It advances the `pc` in the current frame to point to the `imul`, but then it pushes a new frame and positions the `pc` at the entry, `0`. That is also what our invariant predicate, `Inv`, does! So the next instruction encountered by `Inv` is the first instruction of the recursively entered `rfact`, which is annotated with a pre-condition. That pre-condition concludes the generation of a proof obligation. So when do we get back to consider the `imul`? That is, what path handles the case where we actually do the multiplication of the recursively computed $(n-1)!$ and n? The answer is: the path that starts at the `ireturn`! It immediately pops a frame and continues stepping. Therefore, the assertion at the `ireturn` must, apparently, characterize the `pc` and stack in the caller's frame.

This approach violates the appeal of the inductive assertion method. We believe the methodology for handling procedure invocations would be simpler if we had a "big step" semantics, whereby procedure bodies are run in a "single step" and control is then advanced to the next instruction in the byte stream. Such a model can be derived from and proved equivalent to the "little step" semantics, by appropriate restructuring of the definition of `run`. This work remains to be done.

We have studied compositional proof techniques, whereby partial correctness theorems about procedures can be combined to verify sequentially composed procedures. Such techniques are discussed in [17].

13 Conclusion

This paper has demonstrated that inductive-assertion-style proofs can be carried out in an operational semantics framework, without producing a verification condition generator or incurring proof obligations beyond those produced by such a tool. The key insight is that assertions attached to cut points in a program can be propagated by a tail-recursive function to create an alleged invariant. The proof that the alleged invariant is invariant under the state transition function produces the standard verification conditions. The invariance result can then be traded in for a partial correctness result stated in terms of the operational semantics, without requiring the construction of clocks or the counting of instructions.

In this paper, procedure invocation and compositional proof techniques are dealt with only briefly and by reference to [14, ?]. The techniques described leave much to be desired because of the need to characterize the invocation stack as part of the assertions. Much work remains to make this technique feasible.

But its advantages are clear. No verification condition generator need be constructed. Given an operational semantics it is possible, more or less immediately, to perform inductive-assertion-style proofs of partial correctness theorems.

The process of proving the step-wise invariance of the completed assertions "naturally" produces the verification conditions.

This situation is attractive for three reasons. First, writing a verification condition generator for a realistic programming language like JVM bytecode is error-prone. For example, method invocation involves complicated non-syntactic issues like method resolution with respect to the object on which the method is invoked, as well as side-effects to many parts of the state including, possibly, the call frames of both the caller and the callee, the thread table (in the event that a thread is started), the heap (in the event of a synchronized method locking the object upon which it is invoked), and the class table (in the event of dynamic class loading). Coding this all in terms of formula transformation instead of state transformation is difficult. Second, when completed, the semantics of the language is encoded in the VCG process rather than as sentences in a logic. This encoding of the semantics makes it difficult to inspect. In our approach, the semantics is expressed explicitly in the logic so that it can be inspected. Indeed, it is possible to prove theorems about the semantics (not just theorems about programs under the semantics). Finally, realistic VCGs contain simplifiers used to keep the generated proof obligations simple. These simplifiers are just theorems provers and must be trusted. In our approach, only one theorem prover is involved. It must be trusted but that trusted engine derives the verification conditions from the operational semantics and the user-supplied assertions.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.
2. R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, volume XIX, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967.
3. P. Y. Gloess. Imperative program verification in PVS. Technical Report `http://-dept-info.labri.u-bordeaux.fr/~gloess/imperative/index.html`, École Nationale Supérieure Électronique, Informatique et Radiocommunications de Bordeaux, 1999.
4. H. H. Goldstine and J von Neumann. Planning and coding problems for an electronic computing instrument. In *John von Neumann, Collected Works, Volume V.* Pergamon Press, Oxford, 1961.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

6. P. Homeier and D. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.
7. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
8. J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, Boston, MA., 1999.
10. H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.
11. P. Manolios and J S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.
12. J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.
13. J S. Moore. An NQTHM formalization of a small machine. Technical Report ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/nqthm-1992/examples/basic/-small-machine.events, Computational Logic, Inc., May 1991.
14. J S. Moore. Inductive assertions and operational semantics – long version. Technical Report `http://www.cs.utexas.edu/users/moore/publications/trecia/-index.html`, Department of Computer Sciences, University of Texas at Austin, 2003.
15. J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.
16. J S. Moore and G. Porter. The Apprentice challenge. *ACM TOPLAS*, 24(3):1–24, May 2002.
17. S. Ray and J S. Moore. Proof styles in operational semantics. Technical report, Department of Computer Sciences, Univesity of Texas at Austin, (submitted for publication, 2004).
18. A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, England, June 1949.