Inductive Assertion Style
Proofs via Operational Semantics

J Strother Moore
Department of Computer Sciences
The University of Texas at Austin

September, 2006
Demo 1
Our Secret

Pick a traditional mathematical logic, put your effort into building a powerful symbolic manipulation engine for it, and get on with the task of describing and analyzing computing systems.
So Much for History...

This talk is about how to do inductive assertion style proofs with an operational semantic model.

No Floyd-Hoare semantics need be expressed.

No verification condition generator (VCG) need be defined.
Operational Semantics

The *semantics* of the programming language may be given by a function $\text{run}$ which “interprets” a program against some state and determines the “final” state.

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

Here, $\text{step}$ is the single step state transition function.
Conventions

VC1. \( P(s) \rightarrow R(f(s)) \),
VC2. \( R(s) \land t \rightarrow R(g(s)) \), and
VC3. \( R(s) \land \neg t \rightarrow Q(h(s)) \).
We assume the program in $s, \pi$, does not change during execution.

Let $s_0$ be the initial state of program $\pi$.

$p_c(s_0) = \alpha$

Let $s_k$ denote $run(k, s_0)$. 
Formally Stated Correctness Theorems

Total:

\[ \exists k : P(s_0) \rightarrow (Q(s_k) \land pc(s_k) = \gamma). \]

This is sometimes stated without the quantifier as

\[ P(s_0) \rightarrow (Q(\text{run}(\text{clock}(s_0), s_0)) \land \ldots). \]

Partial:

\[ P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k). \]
Advantages of Operational Semantics

They

● are entirely within a logical framework and so permit logical analysis of programs by traditional formal proofs, without introduction of meta-logical transformers (VCGs),

● are generally executable,

● are easily related to implementations,

● allow derivation of language properties,
• may allow derivation of intensional properties (e.g., how many steps a program takes to terminate), and

• they allow verification of system hierarchies (e.g., the CLI stack).
A Disadvantage

Proofs of program properties can be complicated (or at least appear so) because of the presence of the interpreter, the program counter, the entire machine state, and the clock.

The inductive assertion method produces such nice proof obligations!
Conundrum

Can you prove

\[ P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k). \]

directly – where the only heavy-duty proof work is proving the verification conditions?

Do you need a trusted VCG?

Can you make the automatic proof attempt \textit{generate} the standard verification conditions from the operational semantics?
Caveat

The observations I make below are not deep, but I think they have important practical implications:
**Theorem:** \( P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k) \)

**Proof:** Define

\[
Inv(s) \equiv \begin{cases} 
P(s) & \text{if } pc(s) = \alpha \\
R(s) & \text{if } pc(s) = \beta \\
Q(s) & \text{if } pc(s) = \gamma \\
Inv(step(s)) & \text{otherwise}
\end{cases}
\]

(Actually, we assert "\( prog(s) = \pi \)" at \( \alpha, \beta \) and \( \gamma \), but we omit that here by our convention that the program is always \( \pi \).)
**Theorem:** \( P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k) \)

**Proof:** Define

\[
Inv(s) \equiv \begin{cases} 
P(s) & \text{if } pc(s) = \alpha \\
R(s) & \text{if } pc(s) = \beta \\
Q(s) & \text{if } pc(s) = \gamma \\
Inv(step(s)) & \text{otherwise}
\end{cases}
\]

Objection: Is this definition consistent? Yes: Every tail-recursive definition is witnessed by a total function. (Manolios and Moore, 2000)
**Theorem:** \( P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k) \)

**Proof:** Define

\[
Inv(s) \equiv \begin{cases} 
  P(s) & \text{if } pc(s) = \alpha \\
  R(s) & \text{if } pc(s) = \beta \\
  Q(s) & \text{if } pc(s) = \gamma \\
  Inv(step(s)) & \text{otherwise}
\end{cases}
\]

Assume \( Inv(s) \rightarrow Inv(step(s)) \).

We’ll see the proof in a moment.
**Theorem:** \( P(s_0) \land pc(s_k) = \gamma \rightarrow Q(s_k) \)

**Proof:** Define

\[
Inv(s) \equiv \begin{cases} 
P(s) & \text{if } pc(s) = \alpha \\
R(s) & \text{if } pc(s) = \beta \\
Q(s) & \text{if } pc(s) = \gamma \\
Inv(step(s)) & \text{otherwise}
\end{cases}
\]

Thus \( Inv(s_0) \rightarrow Inv(s_k) \).

But \( pc(s_0) = \alpha \) and \( pc(s_k) = \gamma \).

Thus \( Inv(s_0) = P(s_0) \) and \( Inv(s_k) = Q(s_k) \). \( \square \)
Lemma: $\text{Inv}(s) \rightarrow \text{Inv}(\text{step}(s))$

Proof: Consider cases on $pc(s)$ as suggested by def $\text{Inv}$.

\[\text{Inv}(s) \equiv \begin{cases} 
    P(s) & \text{if } pc(s) = \alpha \\
    R(s) & \text{if } pc(s) = \beta \\
    Q(s) & \text{if } pc(s) = \gamma \\
    \text{Inv}(\text{step}(s)) & \text{otherwise}
\end{cases}\]
\( \text{Inv}(s) \rightarrow \text{Inv}(\text{step}(s)) \)

\[
\text{Inv}(s) \equiv \begin{cases} 
    P(s) & \text{if } \text{pc}(s) = \alpha \\
    R(s) & \text{if } \text{pc}(s) = \beta \\
    Q(s) & \text{if } \text{pc}(s) = \gamma \\
    \text{Inv}(\text{step}(s)) & \text{otherwise}
\end{cases}
\]
\[ \text{Inv} \ (s) \rightarrow \text{Inv} \ (\text{step} \ (s)) \]

\textit{case} subgoal generated by expanding \textit{Inv}

\[ pc \ (s) = \alpha: \ \ P \ (s) \rightarrow R \ (f \ (s)) \]

\[ pc \ (s) = \beta: \ \ R \ (s) \wedge t \rightarrow R \ (g \ (s)), \ \text{and} \]
\[ R \ (s) \wedge \neg t \rightarrow Q \ (h(s)) \]

\[ pc \ (s) = \gamma: \ \ \text{step} \ (s) = s \]

\textit{otherwise}: \ \ \text{Inv} \ (s) = \text{Inv} \ (\text{step} \ (s)) \]
Recap: Given the definition of $Inv$, the “natural” proof of

$$Inv(s) \rightarrow Inv(step(s))$$

generates the standard verification conditions

VC1. $P(s) \rightarrow R(f(s))$,

VC2. $R(s) \land t \rightarrow R(g(s))$, and

VC3. $R(s) \land \neg t \rightarrow Q(h(s))$

as subgoals from the operational semantics!

No VCG is necessary. The VCs are simplified as they are generated.
A Demonstration

- Java and the JVM – a program $\pi$ to compute $n/2$
- a pre-existing operational semantics for the JVM
- execution of the operational semantics
- an inductive invariant proof of partial correctness
Java and the JVM

```
pi(246) = 123
```

```
pi(n) = n/2
```

Theorems

```
"pi(246)=123"
```

```
"pi(n)=n/2"
```
JVM Operational Semantics

Our “M6” model is based on an implementation of the J2ME KVM. It executes most J2ME Java programs (except those with significant I/O or floating-point).

M6 supports all CLDC data types, multi-threading, dynamic class loading, class initialization and synchronization via monitors.
We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes. We provide implementations for 21 out of 41 native APIs that appear in Sun’s CLDC API library.

We prove theorems about bytecoded methods with the ACL2 theorem prover.

*This work was supported by a gift from Sun Microsystems.*
Disclaimers about Our JVM Model

Our thread model assumes

• sequential consistency and

• atomicity at the bytecode level.

This inductive assertion work does not exercise the thread model.
Demo 2
Discussion

We did not write a VCG for the JVM.

The VCs were generated directly from the operational semantics by the theorem prover.

Since VCs are generated by proof, the paths explored and the VCs generated are sensitive to the pre-condition specified.

The VCs are simplified (and possibly proved) by the same process.

We did not count instructions or define a “clock
function.”

We did not constrain the inputs so that the program terminated.

We have also handled total correctness via the VCG approach; a decreasing ordinal measure is provided at each cut point. “Clock functions” can be automatically generated and admitted from such proofs.
Primary Citation

Other Examples

Nested loops are handled exactly as by standard VCG methods.

```java
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;
}
```
Recursive methods can be handled.

```java
public static int fact(int n){
    if (n>0)
        return n*fact(n-1);
    else return 1;
}
```

To handle recursive methods we

- modify `run` to terminate upon top-level `return`, and

- add a standard invariant about the shape of the JVM call stack.
Conclusion

If you have

• a theorem prover and

• a formal operational semantics,

you can prove formally stated partial program correctness theorems using inductive assertions without building or verifying a VCG.
Related Work

