Inductive Assertions and Operational Semantics

J Strother Moore

Department of Computer Sciences
The University of Texas at Austin

October 27, 2003

Operational Semantics

– was first formalized by McCarthy in the early 1960s.

Choose some suitably expressive mathematical logic.

A machine state is an object (e.g., n-tuple) in the logic, which typically includes programs.

A program is an object (e.g., sequence of instructions).

The language semantics is given by a function on states (e.g., an interpreter).

This function is defined (axiomatized) within the logic.

A program property is stated as a formula (e.g., "there exists a k such that running the interpreter k steps starting from s_0 produces a state, s_k, with the desired property").

A program property may be a theorem and may be proved directly in the logic.

To prove program properties mechanically one needs a theorem prover for the logic.

The Inductive Assertion Method

– was first formalized by Floyd and Hoare in the mid 1960s.

Program text is annotated with assertion formulas written in a given logic.

These assertions are sometimes called the *program specification*.

Semantics is given by a process that transforms an annotated program into a set of proof obligations.

Proof obligations are formulas in the logic.

Often, these proof obligations are called *verification conditions* and the process is called a *verification condition generator* or vcg.

If the proof obligations are theorems, the program is said to *satisfy* its specification.

The *satisfies* statement is not a formula in the logic.

To prove such program specifications mechanically one needs both the verification condition generator and a theorem prover for the logic.

Program logics were proposed by Hoare, Dijkstra and others to unify this story, but practically speaking the method is decomposed into two steps:

- crawl over the annotated program text to generate formulas (possibly simplifying the intermediate formulas as you go), and then

- submit the generated proof obligations to a theorem prover.

Comparisons

An operational semantics is

- generally *executable* (hence, dual use),
- more easily validated against implementations (e.g., compilers, etc),
- permits logical (rather than *meta*-logical) analysis of semantics, and thus
- enables verification of system hierarchies.

The inductive assertion method

- factors out static analysis so the user can focus on algorithmic invariants, thus

- specific program proofs are often simpler.

The Topic of this Talk

It is possible to unify these two approaches entirely within the operational semantics framework.

Given a formal operational semantics and a theorem prover for the logic, one can apply the inductive assertion method directly.

No extra-logical machinery (e.g., vcg, wp, etc) is necessary.

Caveat

As far as I know, this observation has never been made before, though it is not deep.

All previous formal work that connects vcg to operational semantics did so by defining and then formally proving the correctness of a vcg first!

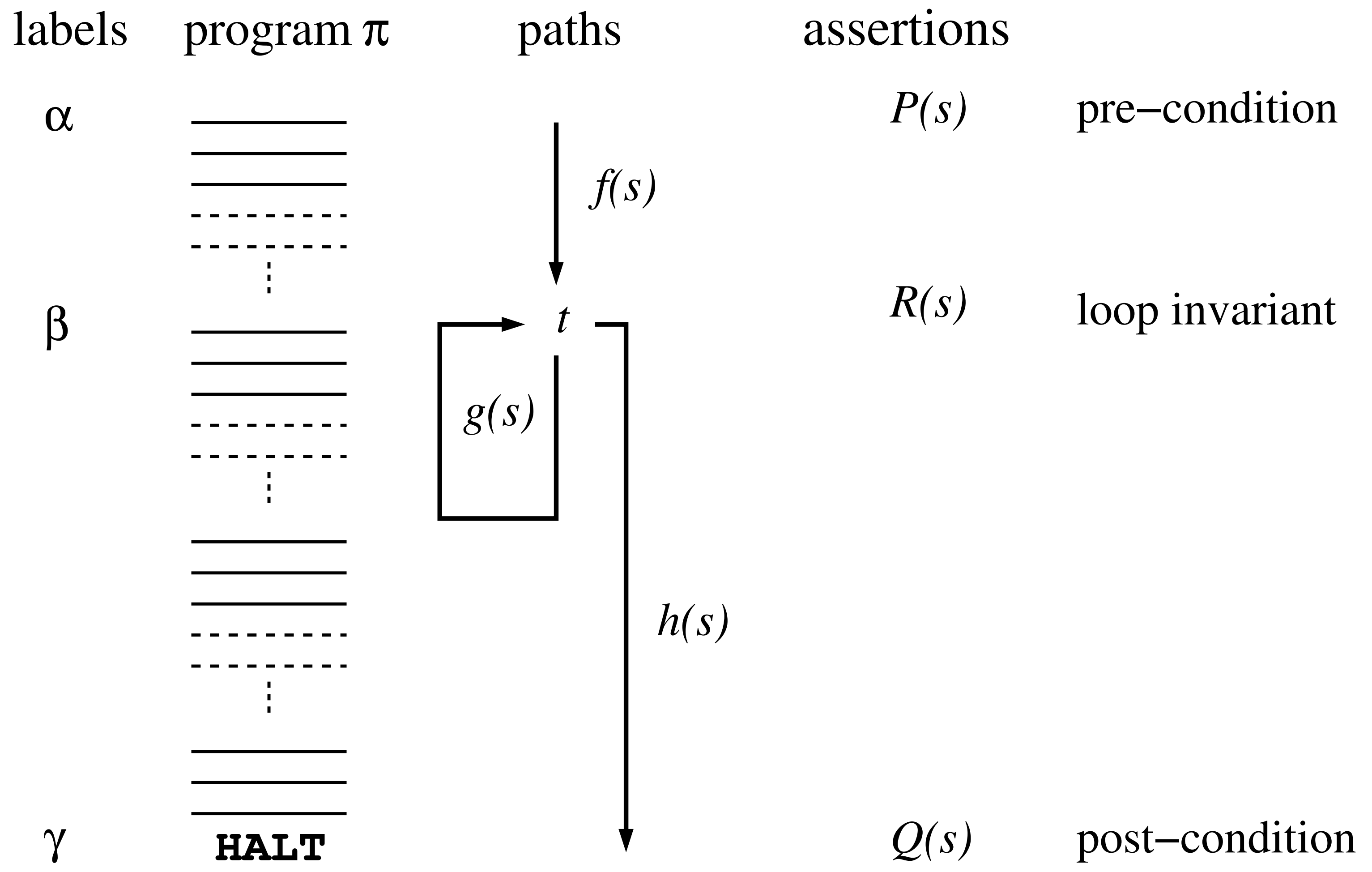
The classic work on inductive assertions frequently referred to operational semantics to establish the correctness of the vcg method.

It is known that the two are mathematically equivalent.

There is no theoretical contribution here.

The contribution is entirely a practical one.

Program π


labels
program π
paths
assertions
α
β
γ
HALT
f(s)
t
g(s)
h(s)
P(s)
pre–condition
R(s)
loop invariant
Q(s)
post–condition

Operational Semantics

A *state*, s, is typically an object in the logic that includes such components as:

- pc – a program counter
- program – a representation of the code
- memory – mapping from program variables to values

Operational Semantics – Continued

The *state transition* function, $step$, maps from states to states by "executing" the instruction at the current pc.

The *semantics* of the programming language is given by a function run which steps an initial state some number of times or through some given input.

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

Conventions

Let s_0 be a state initialized for the program of interest:

- $prog(s_0) = \pi \;\wedge\; pc(s_0) = \alpha$

Let s_k denote $run\,(k, s_0)$.

Formally Stated Correctness Theorems

Total:

$$\exists k : P\,(s_0) \to Q(s_k).$$

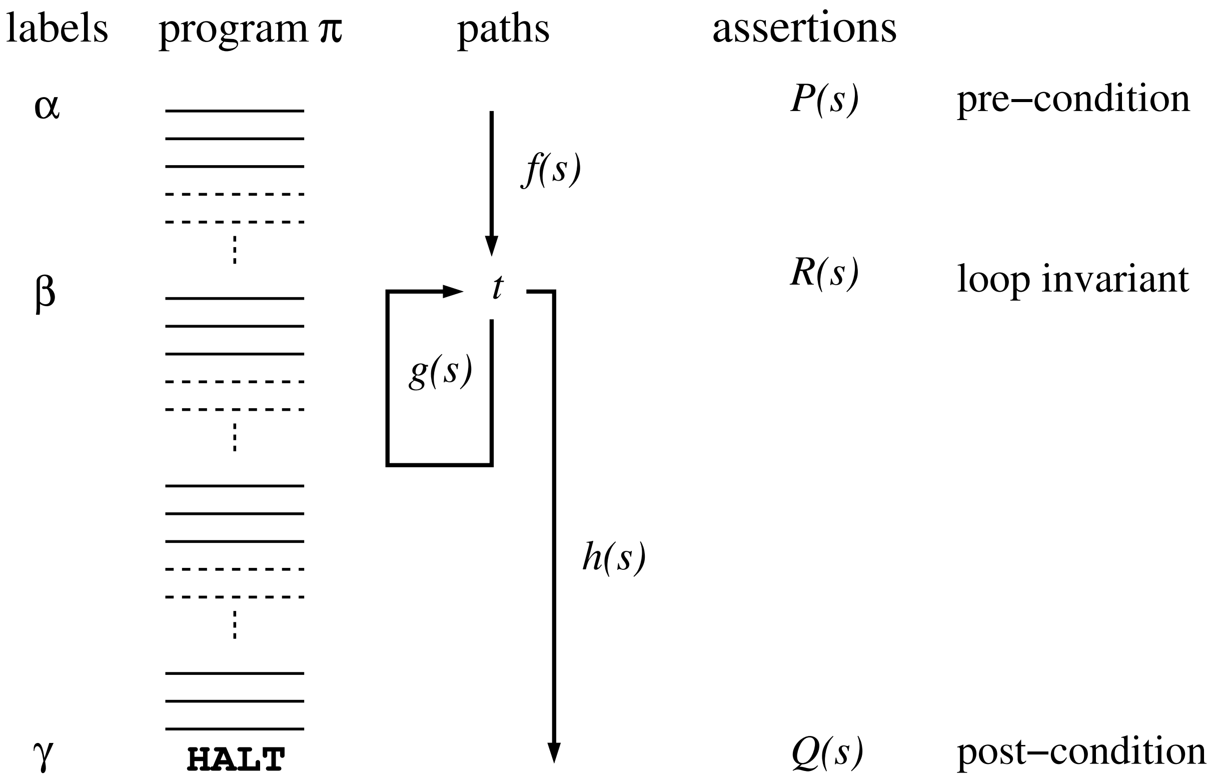
This is sometimes stated without the quantifier as

$$P\,(s_0) \to Q(run\,(clock\,(s_0), s_0)).$$

Partial:

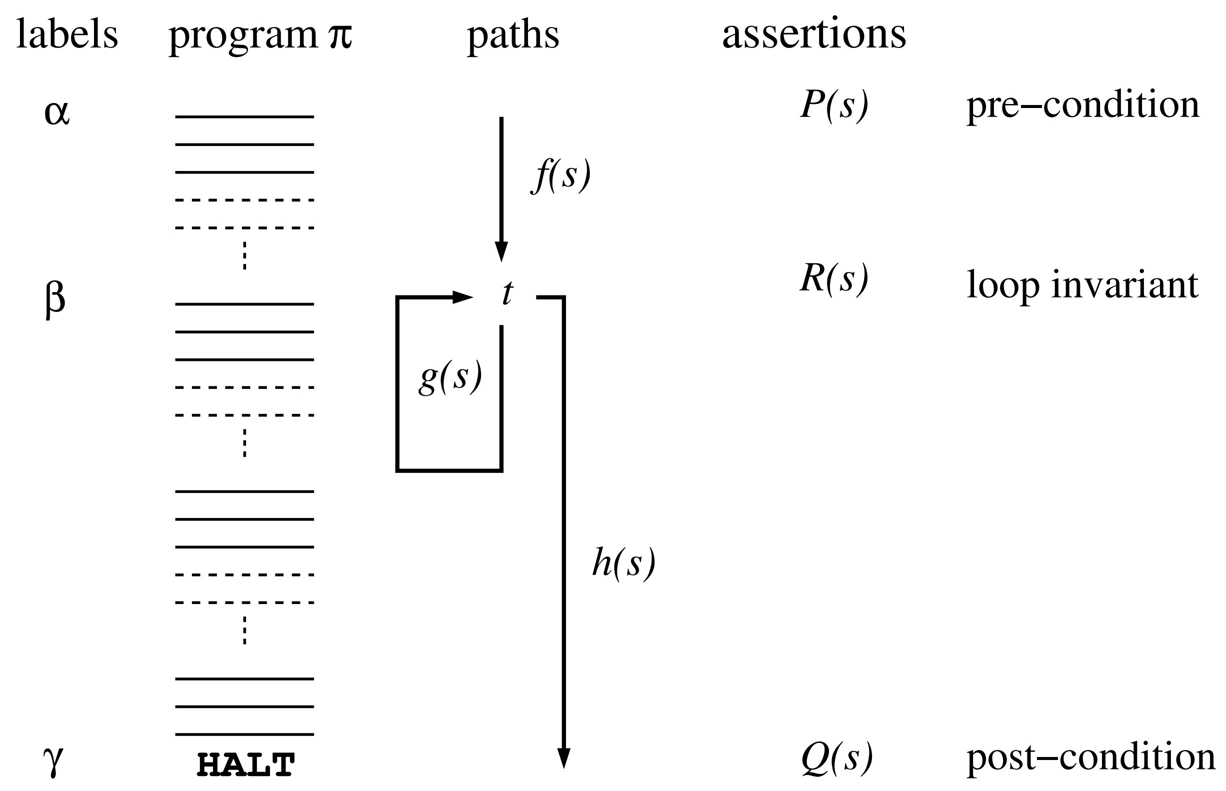
$$P\,(s_0) \;\wedge\; pc(s_k) = \gamma \to Q(s_k).$$

Partial Correctness of Program π



$$P\,(s_0)\ \wedge\ pc(s_k) = \gamma \rightarrow Q(s_k).$$

Verification Conditions



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

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

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

Question

Can you prove

Theorem:

$$P\left(s_0\right) \wedge pc\left(s_k\right) = \gamma \to Q\left(s_k\right).$$

from:

VC1. $P\left(s\right) \to R\left(f\left(s\right)\right)$,
VC2. $R\left(s\right) \wedge t \to R\left(g\left(s\right)\right)$, and
VC3. $R\left(s\right) \wedge \neg t \to Q\left(h(s)\right)$?

Theorem: $P(s_0) \wedge pc(s_k) = \gamma \rightarrow Q(s_k)$

Proof: Define

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

Theorem: $P(s_0) \wedge pc(s_k) = \gamma \rightarrow Q(s_k)$

Proof: Define

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

Objection: Is it consistent? Yes: Every tail-recursive definition is witnessed by a total function. (Manolios and Moore, 2000)

Theorem: $P(s_0) \wedge pc(s_k) = \gamma \rightarrow Q(s_k)$

Proof: Define

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

It follows that

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

We'll see the proof in a moment.

Theorem: $P(s_0) \wedge pc(s_k) = \gamma \rightarrow Q(s_k)$

Proof: Define

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

Thus

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

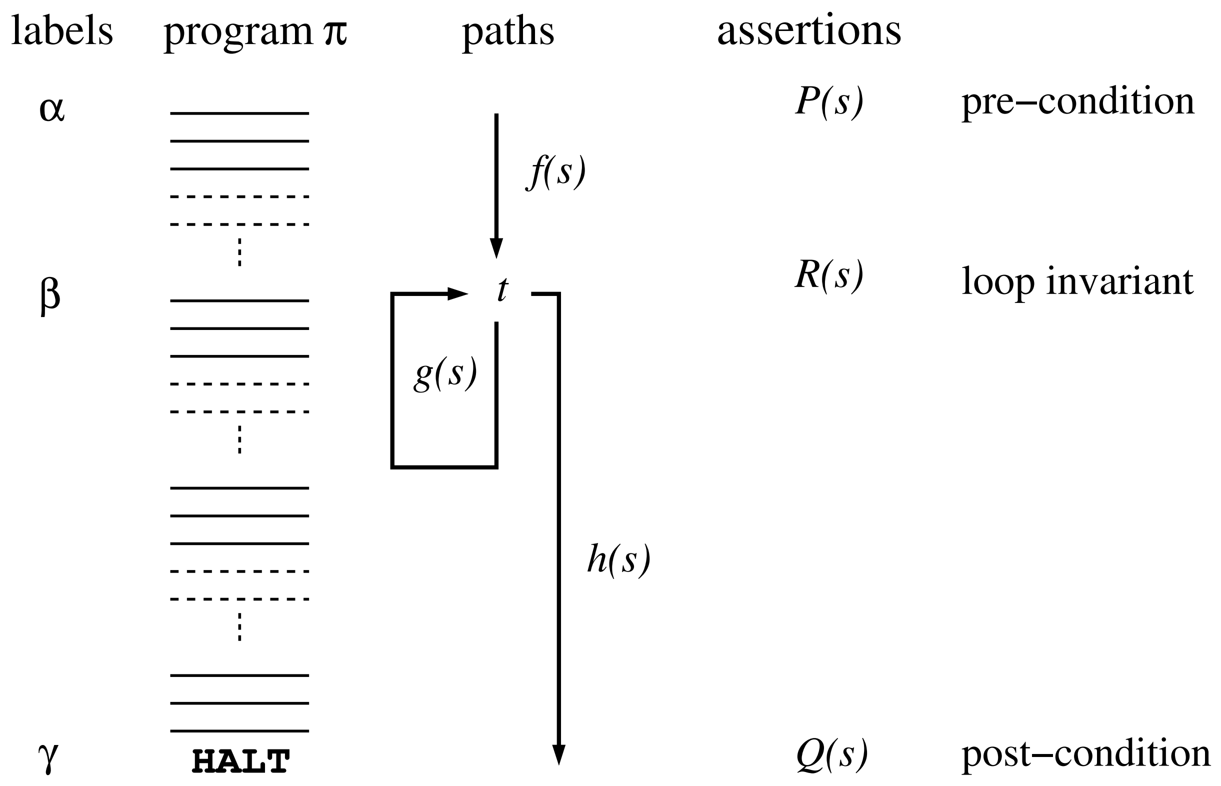
□

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

Proof: Immediate from def Inv and VC1–VC3.

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

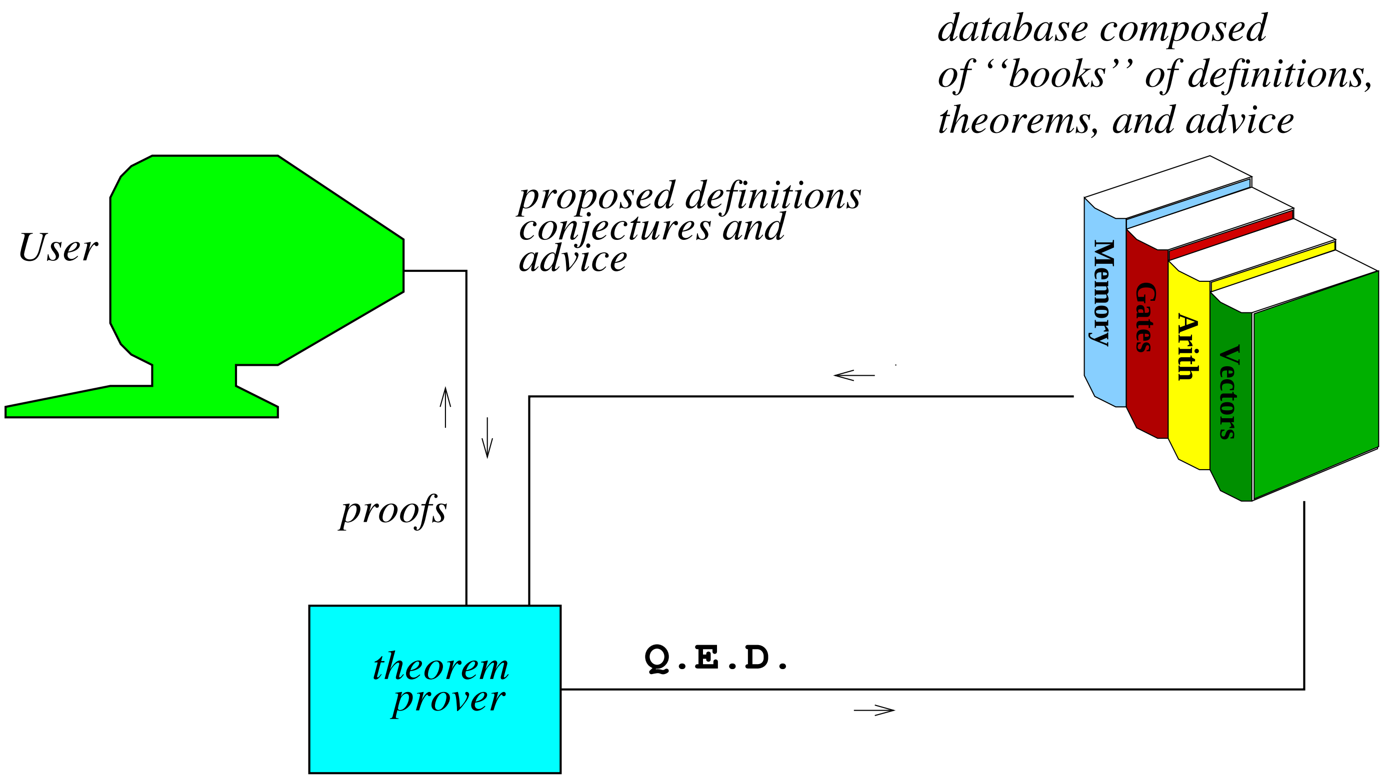
VC1. $P\,(s) \rightarrow R\,(f\,(s))$,
VC2. $R\,(s) \wedge t \rightarrow R\,(g\,(s))$, and
VC3. $R\,(s) \wedge \neg t \rightarrow Q\,(h(s))$
$\square$


labels
program π
paths
assertions
α
β
γ
HALT
f(s)
t
g(s)
h(s)
P(s)
pre−condition
R(s)
loop invariant
Q(s)
post−condition

A Demonstration

- the ACL2 theorem prover
- a pre-existing operational semantics for the JVM
- a bytecode program to compute $n/2$
- a direct proof of total correctness (with clock)
- an inductive invariant proof of partial correctness (without clock)

ACL2: A Computational Logic for Applicative Common Lisp



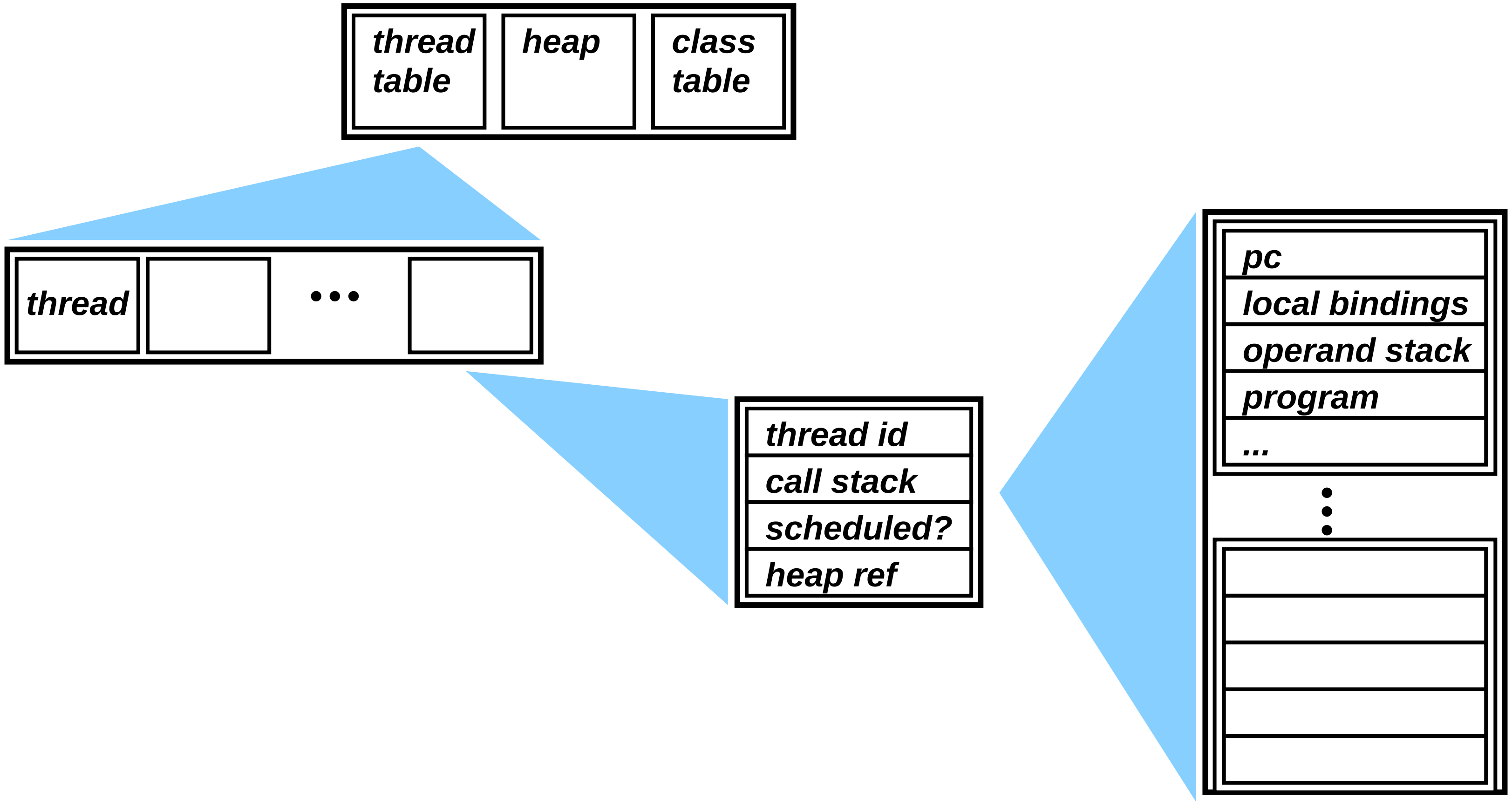
Demo 1

Operational Semantics

```
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched)
           (step (car sched) s))))
```

We have formalized the JVM in ACL2, i.e., defined an executable JVM bytecode interpreter in Lisp. We have tools to translate Java class files to ACL2 objects representing JVM states and we can run them.

Our JVM State: $< tt, hp, ct >$



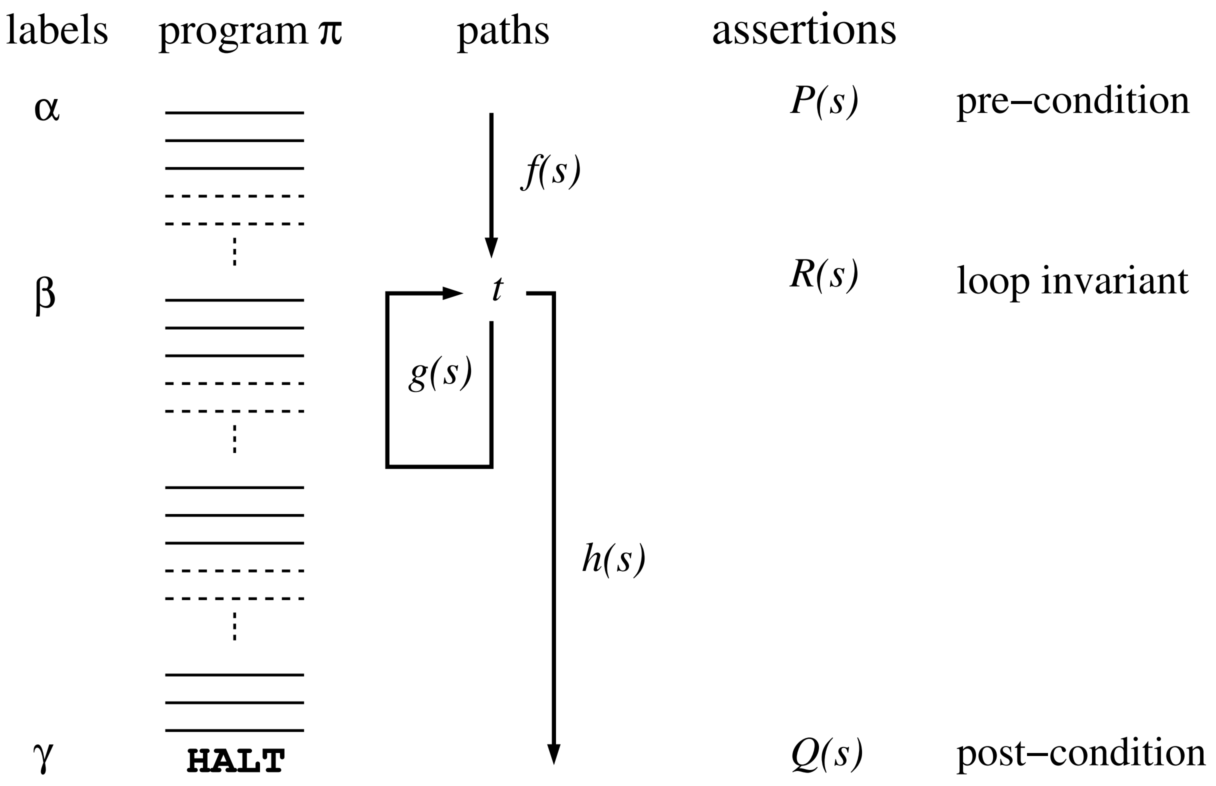
Demo 2

Code Proofs

Given an operational model, a common way to prove code correct is to do *direct total correctness* proofs, using *clock functions* to characterize how long the program runs.

Demo 3

Partial Correctness



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

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

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

Demo 4

How can we prove this from the VCs?

```
(defthm Partial-Correctness-of-Program-Pi
  (implies
   (and (equal n (n th s))
        (in-pi th s)
        (equal (pc (top-frame th s)) 0)
        (<= 0 n)
        (mono-threadedp th k)
        (equal (pc (top-frame th (run k s))) 17))
   (and (evenp n)
        (equal (a th (run k s))
               (/ n 2)))))
```

Demo 5

Discussion

We did not write a VCG for the JVM.

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.

Sandip Ray (forthcoming paper) shows how to mix the “clock” approach with the inductive assertion approach.

Other Examples

Nested loops are handled exactly as by standard VCG methods.

```
public static int tfact(int n){  /* Factorial by repeated addition.     */
    int i = 1;                   /* Verified using inductive assertions */
    int b = 1;                   /* by Alan Turing, 1949.               */
    while (i<=n){
        int j = 1;
        int a = b;
        while (j < i) {
            b = a+b;
            j++;
        };
        i++;
    };
    return b;
}
```

Recursive methods can be handled.

```
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

P. Y. Gloess, "Imperative Program Verification in PVS," École Nationale Supérieure Électronique, Informatique et Radiocommunications de Bordeaux, 1999.

P. Homeier and D. Martin, "A Mechanically Verified Verification Condition Generator," *The Computer Journal*, **38**(2), pp. 131–141, July 1995.

P. Manolios and J Moore, "Partial Functions in ACL2," *JAR* 2003 (to appear).