Theorem Provers as High Assurance Programming Environments

J Strother Moore

Department of Computer Sciences
University of Texas at Austin

Thesis

High-assurance production-quality special-purpose software analysis tools can often be conveniently built within theorem proving environments, by

- defining the semantics in the mechanized logic, and
- configuring the proof-engine appropriately.

and *Voila!*:

- an execution engine
- a symbolic execution engine
- a verification system
- a verification condition generator
- ...

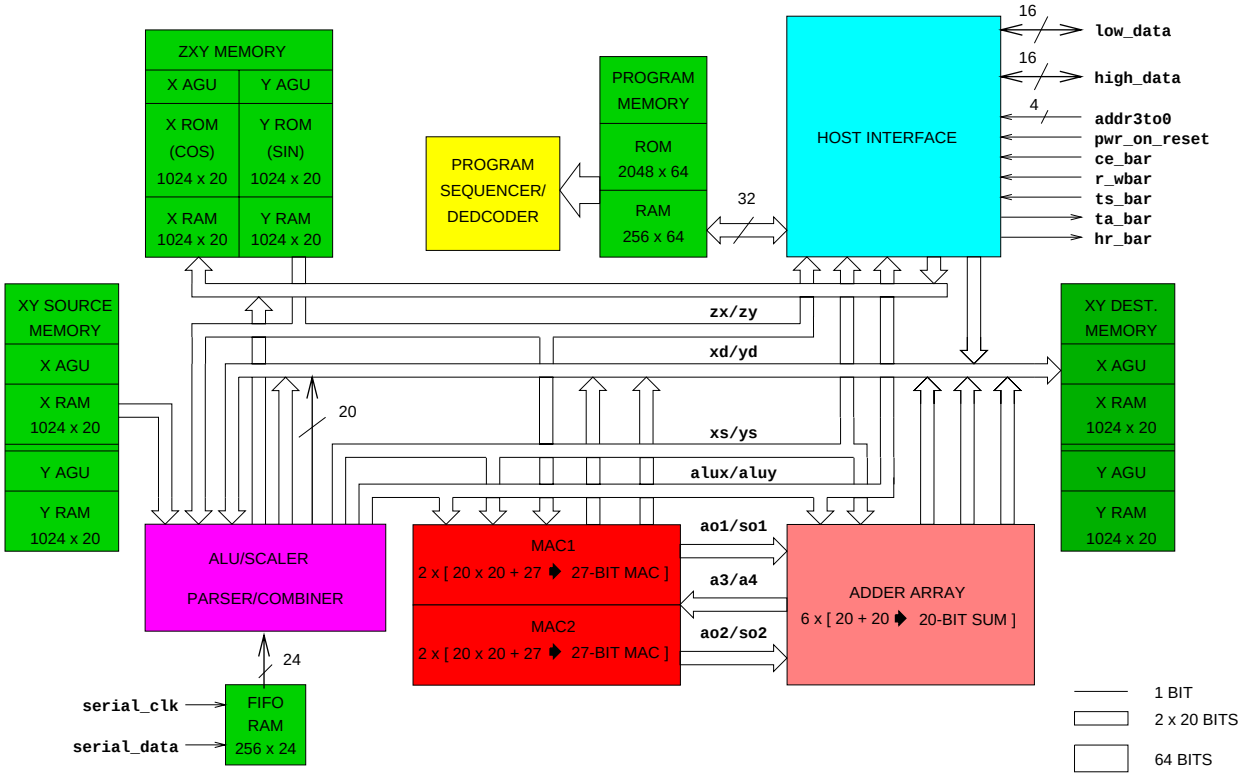
Some Proof Systems Often So Used

ACL2 – a Boyer-Moore-style system for functional Common Lisp (Kaufmann and Moore, *et al*)

HOL – a tactic-based prover for higher order logic (Gordon, *et al*)

Maude – a rewriting logic with a very fast rewrite engine (Meseguer, *et al*)

Motorola CAP DSP

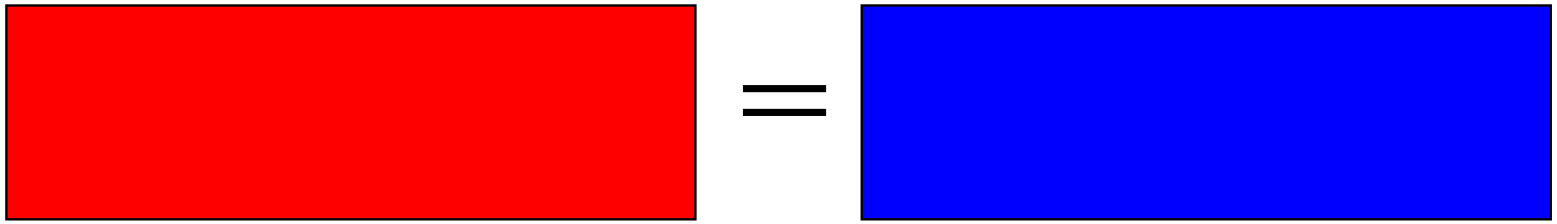


Modeled in ACL2 by Brock.

ROM containing 50 microcoded DSP algorithms

Pipelined microarchitecture

Sequential microcode ISA



(if no hazards)

The model was bit- and cycle-accurate.

The ISA model was proved equivalent to the microarchitecture provided the microcode to be executed avoided a set of precisely defined pipeline hazards.

Microcoded DSP applications were proved correct mechanically.

The ISA model executed several times faster than Motorola's SPW model.

Rockwell-Collins / aJile Systems JEM1

The world's first silicon Java Virtual Machine was first modeled in ACL2. (Greve, Hardin, and Wilding)

The formal microarchitecture model was executable.

It replaced the C model in the test bench upon which Java programs were executed.

The formal model executed at about 90% of the C model.

AMD Athlon Floating Point

The Athlon floating-point unit was modeled in ACL2 by mechanically translating the RTL to ACL2 (Flatau and Russinoff).

80 million floating point test vectors were run through the ACL2 model of FSQRT and were bit-equivalent to those produced by AMD's RTL simulator.

All elementary fpu operations (FADD, FSUB, FMUL, FDIV, FSQRT) were verified IEEE compliant — after fixing four bugs found by the proof attempt.

The same procedure was applied to the AMD Opteron (64-bit).

The ACL2 models gain credibility via their dual-use.

JVM in Maude

A model of the JVM is written in Maude by Meseguer, Farzan, Cheng, and Rosu.

The model exploits Maude's search strategy to give semantics to concurrency.

The Maude model of the JVM provides a symbolic execution capability for concurrent JVM programs.

Accellera PSL in HOL

PSL (formerly IBM's Sugar 2.0) was modeled in HOL by Gordon, Hurd, and Slind.

HOL can then be used to provide an execution engine for the Language Reference Manual.

"Goal is to show formal semantics is not just documentation."

Outline

The rest of this talk will be a mix of ACL2 demonstration and an explanation of how we used ACL2 to build a *verification condition generator* for the JVM.

- classic ACL2 example
- JVM operational semantics
- VCG idea
- inductive assertion proof of a JVM bytecoded method

"ACL2" stands for

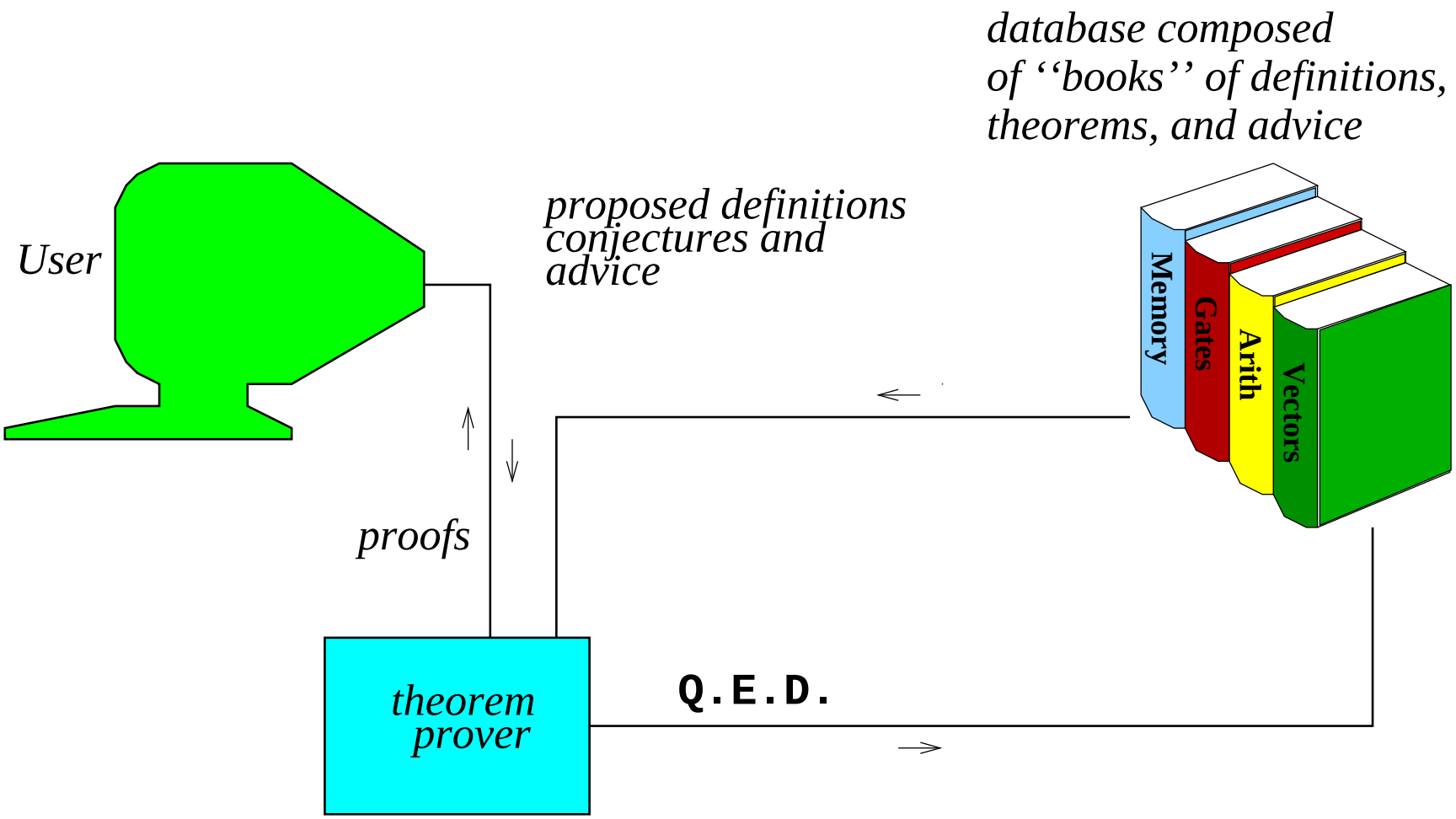
A Computational **L**ogic

for

Applicative **C**ommon **L**isp

Matt Kaufmann and J Moore

See
http://www.cs.utexas.edu/users/moore/acl2


database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.

ACL2 Demo 1

The abstractions of Java are nicely captured by the Java Virtual Machine (JVM).

We verify Java programs by verifying the bytecode produced by the Java

We formalize the JVM with an operational semantics in the ACL2 logic.

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 data types (except floats), 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 is supported by a gift from Sun Microsystems.

Disclaimers about Our JVM Model

Our thread model assumes

- sequential consistency and
- atomicity at the bytecode level.

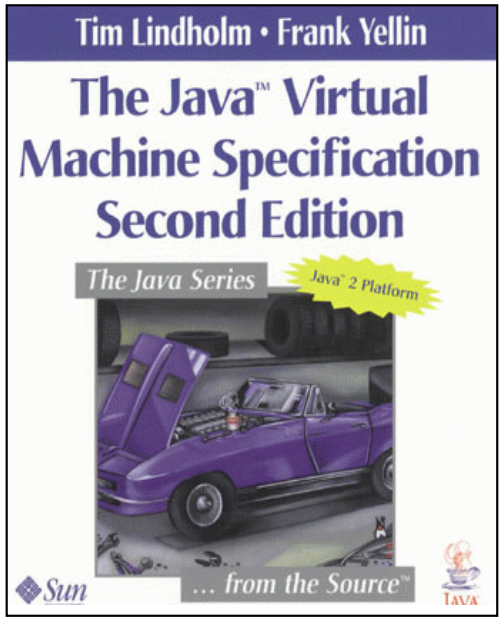
Java and the JVM

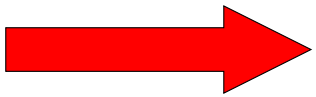
```
class Demo {

  public static int fact(int n){
    if (n>0) {return n*fact(n-1);}
    else return 1;
  }
  public static void main(String[] args){
    int n = Integer.parseInt(args[0], 10);
    System.out.println(fact(n));
    return;
  }}
```

Demo.java

Translating the JVM Spec into ACL2





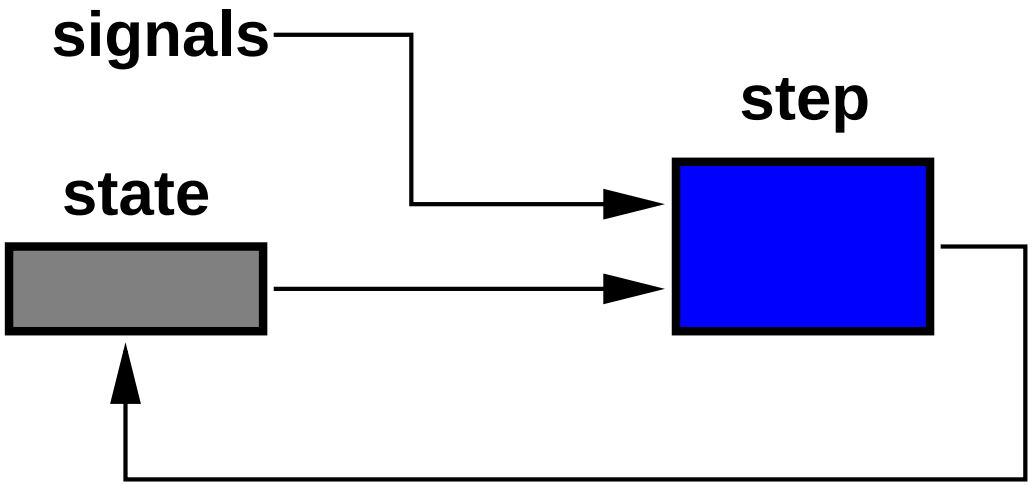
```
; JVM in ACL2


(defun make-state (tt hp ct)
  ...)

(defun step (th s)
  ...)

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

We define a Lisp interpreter for bytecode.




```
(defun run (signals state)
  (if (endp signals)
      state
      (run (cdr signals)
           (step (car signals) state))))
```

The JVM Spec from Sun

iload_0

Operation

Load `int` from local variable 0

Format

iload_0

Form

26 (0x1a)

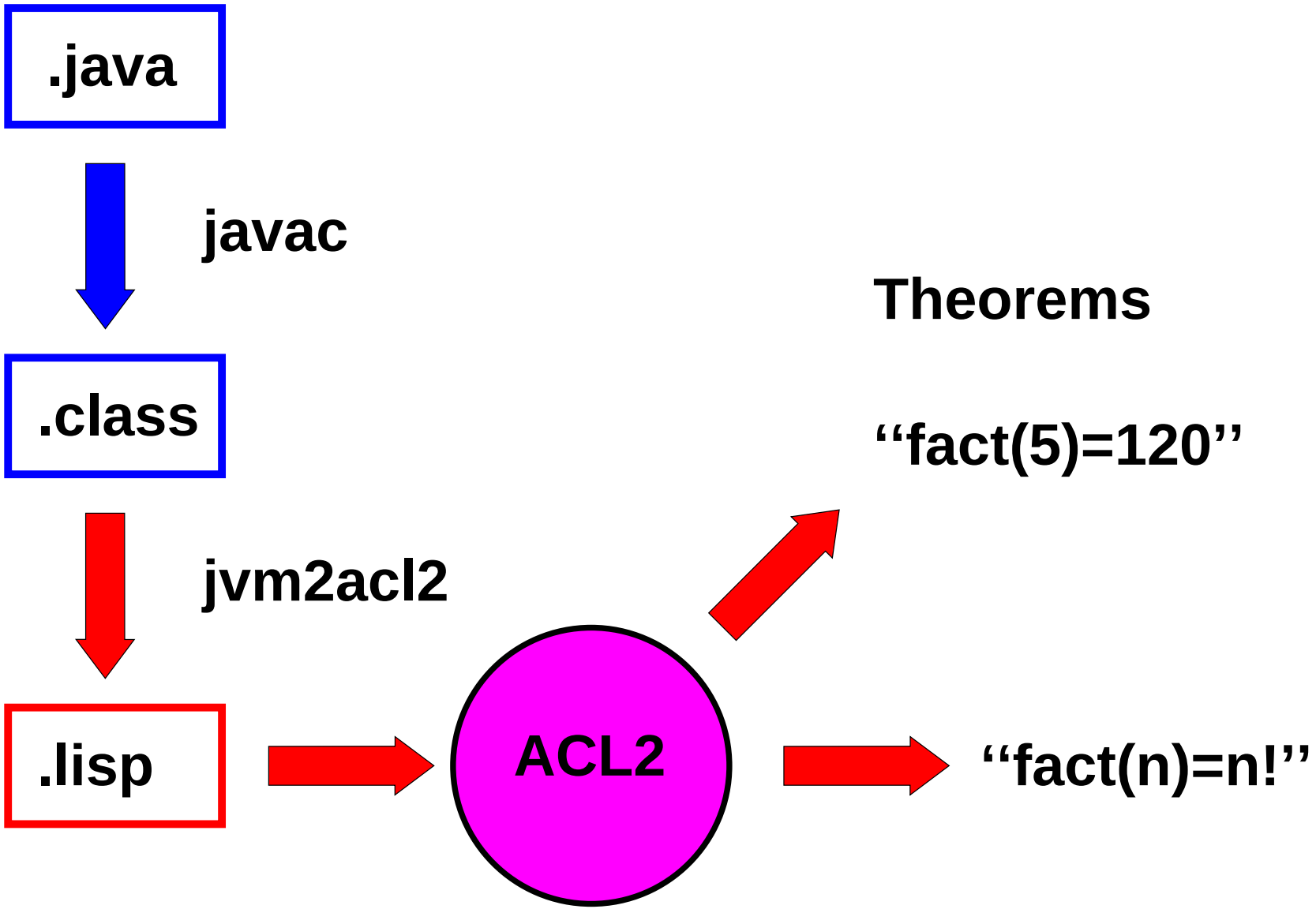
Operand Stack

... $\Rightarrow$ ..., value

Description

The local variable at 0 must contain an `int`. The value of the local variable at 0 is pushed onto the operand stack.

Note: `ILOAD_0`, ... `ILOAD_3` are one-byte specializations of the more general three-byte `ILOAD` n instruction.


.java
javac
.class
jvm2acl2
.lisp
ACL2
Theorems
“fact(5)=120”
“fact(n)=n!”

ACL2 Demo 2

This Model Is Executable

We define `(jvm-Demo param)` to

- build a JVM state poised to invoke the `main` method of class `Demo` on command line `param`,
- use `simple-run` to step that state to completion, and
- print some results.

ACL2 Demo 3

We get execution speeds of about 1000 bytecodes/sec on a 728 MHz processor.

We suspect this could be increased $\times 100$ using ACL2 optimization features.

But This Model is Formal

It is possible to *prove* theorems about this JVM model.

Let's prove that `fact` returns the low-order 32 bits of the mathematical factorial.

```
(defthm fact-is-correct
 ∃ k
  (implies
   (poised-to-invoke-fact s n)
   (equal (simple-run s k)
          (state-set-pc (+ 3 (pc s))
            (pushStack (int-fix (! n))
                       (popStack s))))))
```

```
(defthm fact-is-correct

    (implies
     (poised-to-invoke-fact s n)
     (equal (simple-run s (fact-clock n))
            (state-set-pc (+ 3 (pc s))
              (pushStack (int-fix (! n))
                         (popStack s))))))
```

ACL2 Demo 4

Such proofs are sometimes called *direct* or *clock-style* proofs because they proceed by direct appeal to the operational semantics and (informally) "by induction on the number of steps."

Uses of the Model

execution environment

symbolic execution environment

program verification via operational semantics

analysis of the model (e.g., correctness of bytecode verifier, class loader, etc.)

inductive assertion proof tool (vcg + theorem prover)

Conventions

Let π be a program we want to verify, with entry pc α and exit pc γ.

Let P and Q be the pre- and post-conditions for π.

Let s_0 be a state initialized to run π
i.e., $prog(s_0) = \pi \ \wedge \ pc(s_0) = \alpha$

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

Formally Stated Correctness Theorems

Total:

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

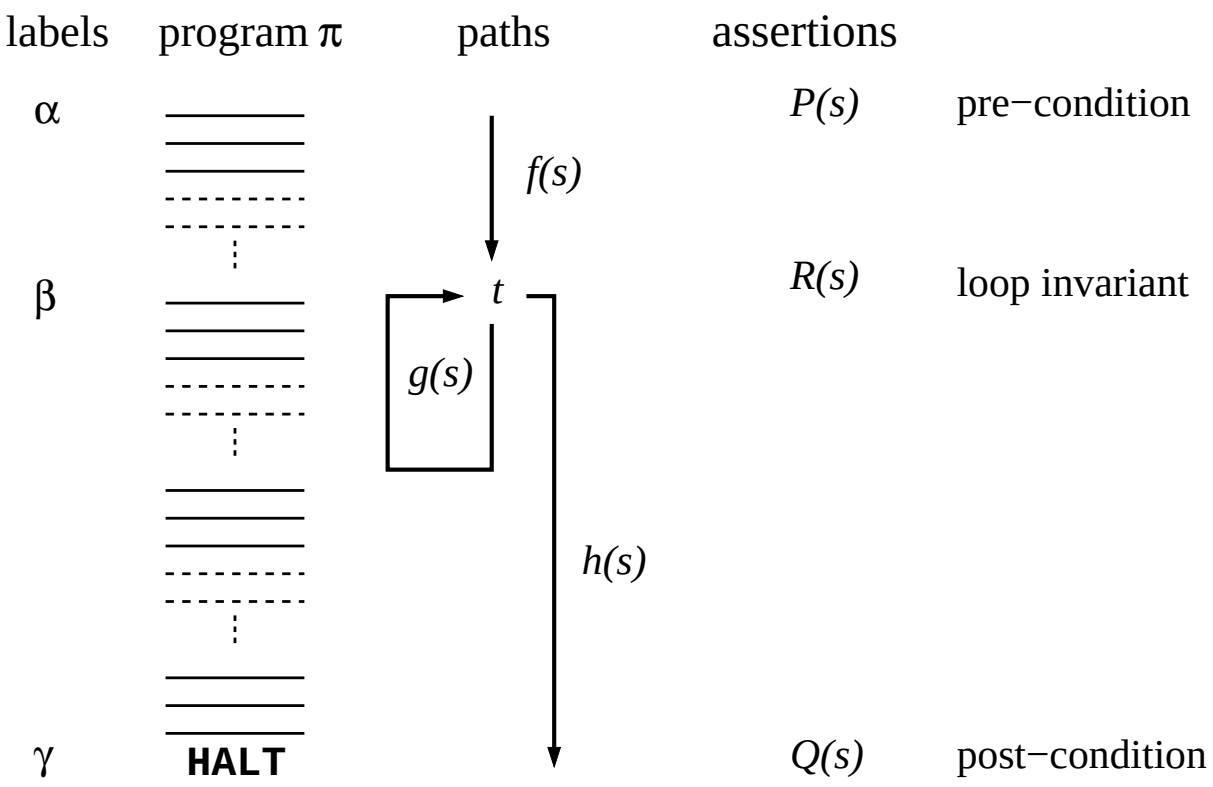
or without quantifier:

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

Partial:

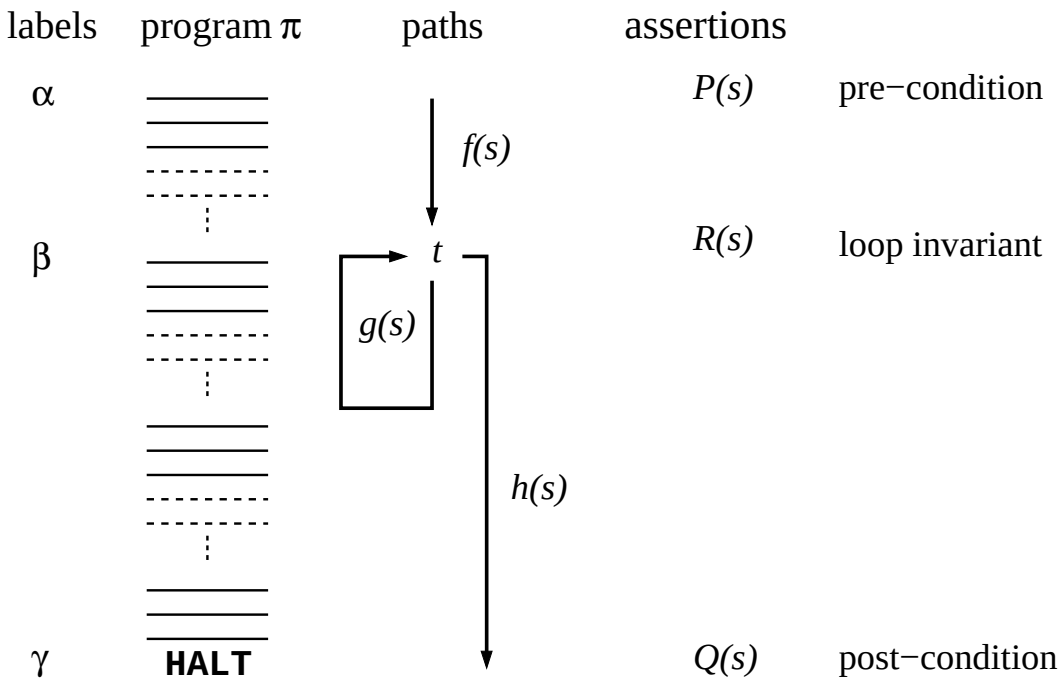
$$P\,(s_0)\ \wedge\ pc(s_k) = \gamma \rightarrow Q(s_k).$$

Partial Correctness of Program π



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

Verification Conditions



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

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

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

Can you prove

Theorem:

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

using operational semantics, by proving

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

without writing a VCG?

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

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

if VC1–VC3 hold.

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

Proof: Define

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

The attempt to prove

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

will *generate* VC1–VC3 (Moore, 2003).

ACL2 Demo 5

What just happened?

We took

- a theorem prover and
- a formal operational semantics

and did an VCG-style proof *without writing a VCG*!

A VCG for the JVM modeled at this level of detail would be very hard to get right!

This method of generating VCs allows the invariants to participate in the control flow exploration.

This method of generating VCs rationalizes the universal mix of VCG and on-the-fly simplification.

Inductive assertion proofs can be mixed direct operational semantics proofs.

Back to the Future?

The idea that theorem proving environments can be used as programming environments is not new.

Prolog evolved in precisely this way in the resolution theorem proving communities of the 1970s.

Acknowledgements

Thanks to Bishop Brock, Rich Cohen, Warren Hunt, Robert Krug, *Hanbing Liu*, Matt Kaufmann, Pete Manolios, George Porter, Sandip Ray, and Rob Sumners, plus dozens of other Nqthm/ACL2 users.

Thank you for listening.

J Moore
Austin, Texas / April, 2004