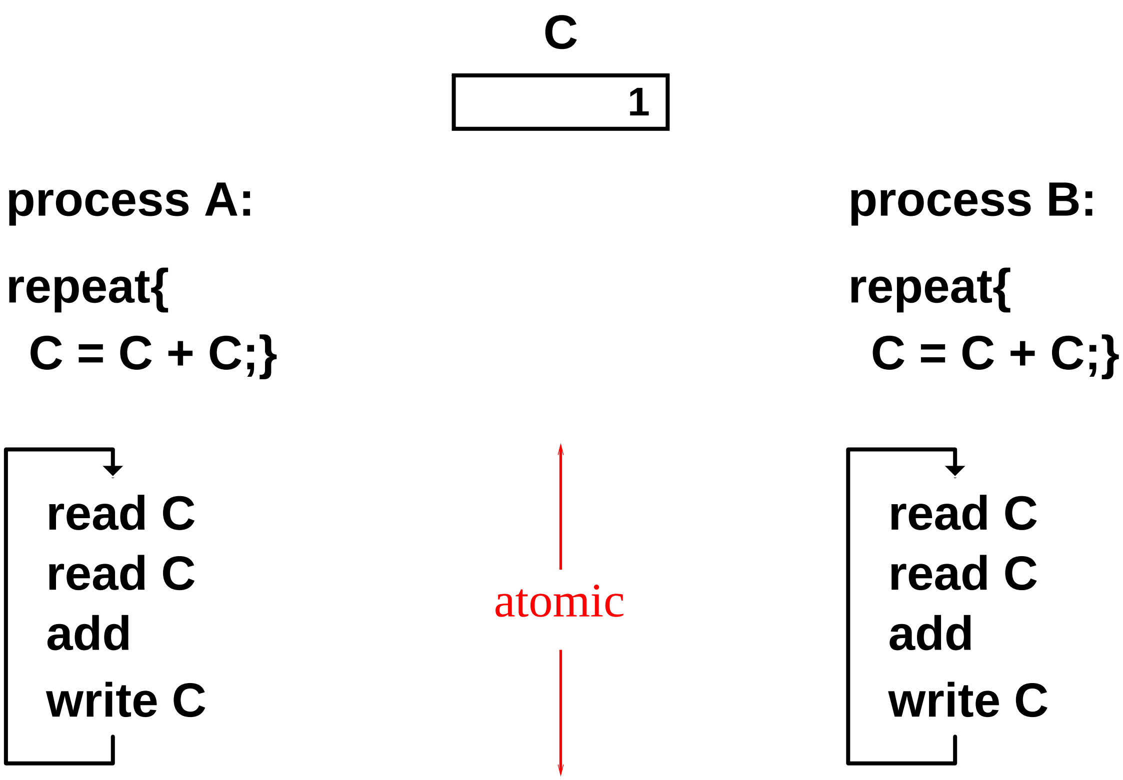
Little (but Hard) Theorems About Big Systems: Some Case Studies

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

Marktoberdorf Summer School 2004

Lecture 5

An Entertaining Puzzle: The Thread Game



Theorem? For every positive integer n there is an interleaving of A and B steps that produces $C = n$.

Thesis

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 compiler.

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.

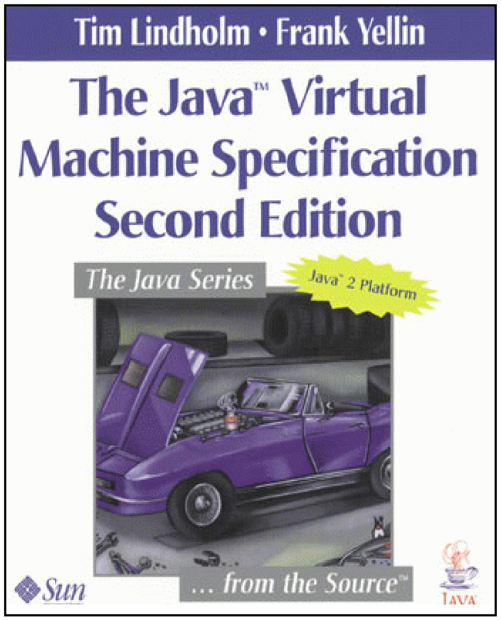
Java and the JVM

```
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



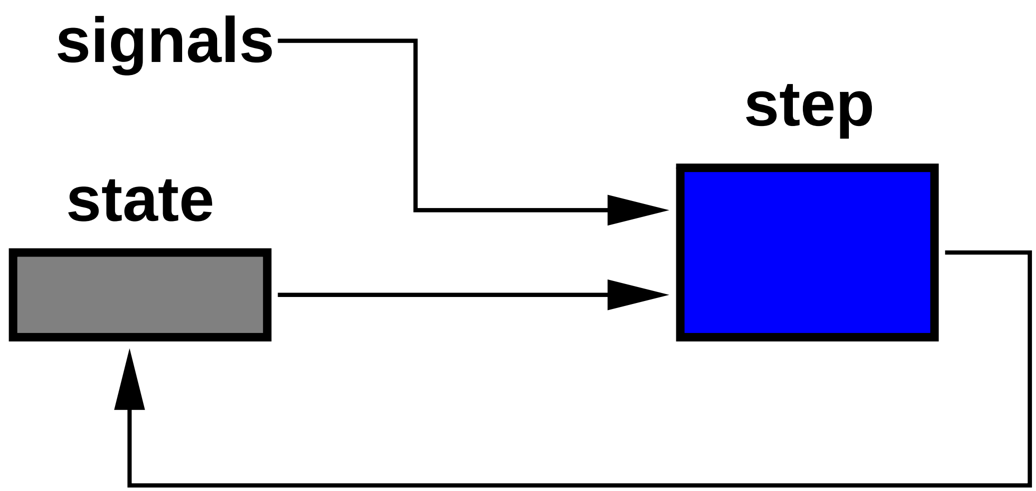

```
; JVM in ACL2


(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)

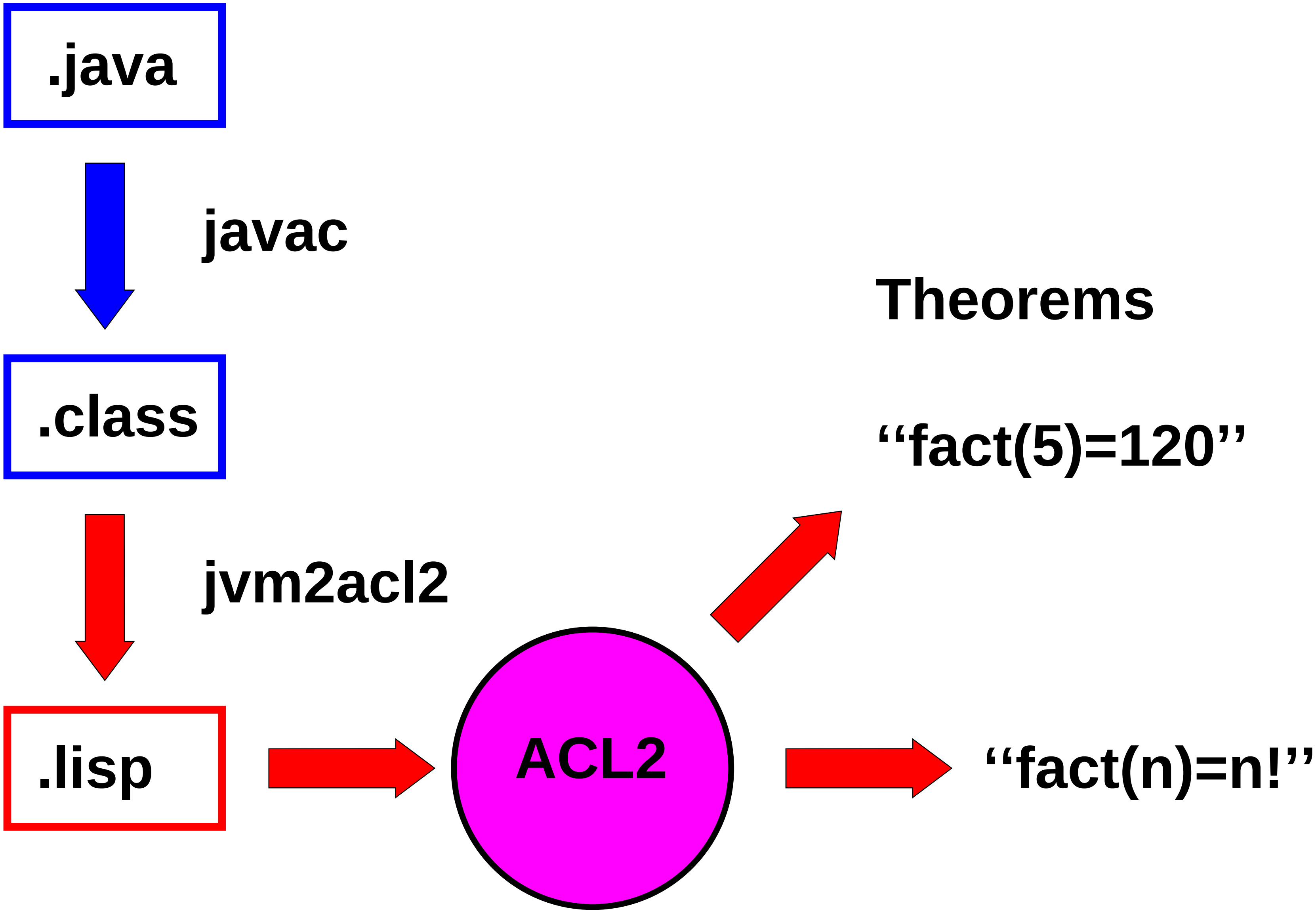
Operand Stack

... $\Rightarrow$ ..., 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

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

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

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.”

```
(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))))))
```

A Precise Informal Notation

We proved that

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

returns

```
(int-fix (! n))
```

We can also prove that

```
class IterativeDemo {
  public static int ifact(int n){
    int temp = 1;
    while (0<n) {
        temp = n*temp;
        n = n-1;}
    return temp;
  } }
```

returns

```
(int-fix (! n))
```

Changing the Heap

```
class Cons {
    int car;
    Object cdr;
    public static Cons cons(int x, Object y){
        Cons c = new Cons();
        c.car = x;
        c.cdr = y;
        return c; } }
```

```
class ListProc extends Cons {
 public static Cons insert(int e,Object x){
  if (x==null)
     {return cons(e,x);}
  else if (e <= ((Cons)x).car)
     {return cons(e,x);}
  else
     return
     cons(((Cons)x).car,
          insert(e,((Cons)x).cdr)); }
```

```
public static Object isort(Object x){
 if (x==null)
    {return x;}
 else return insert(((Cons)x).car,
                  isort(((Cons)x).cdr)); } }
```

Let `deref*` be the function that chases references through the heap (recursively) and constructs the tree represented.

Suppose `isort` is invoked on x_0 in heap h_0 and returns x_1 in heap h_1.

Let α_0 be (`deref*` x_0 h_0), i.e., the list of elements represented by the object x_0 in h_0.

Let α_1 be (`deref*` x_1 h_1).

Then α_1 is an ordered permutation of α_0.

Basic Proof Structure

Lemma 1: Prove that executing the byte code produces a state transformation decribed by a given ACL2 function, i.e., that the `isort` method produces a state change that, modulo `deref*`, is the same as the ACL2 `isort` function.

Lemma 2: Prove that the ACL2 function satisfies the requirements, i.e., produces an ordered permutation.

We Can Prove Partial Correctness Theorems

The proofs mentioned above characterize the number of steps the computations take.

They are "total correctness" theorems.

We can prove partial correctness theorems about non-terminating programs.

The operational semantics can be used directly to do a Floyd-Hoare-style proof.

ACL2 Demo

What just happened?

We took

- a theorem prover and
- a formal operational semantics

and did an inductive assertion proof without adopting a "program logic" or implementing a predicate transformer for Java.

Random Remarks

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

This method rationalizes the universal mix of predicate transformation and on-the-fly simplification.

Inductive assertion proofs can be mixed with direct operational semantics proofs.

A Class Involving Multiple Threads

```
class Container {
    public int counter; }

class Job extends Thread {
    Container objref;

    public void setref(Container o) {
        objref = o; }
```

```
public Job incr () {
    synchronized(objref) {
     objref.counter = objref.counter + 1; }
    return this; }

public void run() {
    for (;;) {incr(); } } }
```

```
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {Job job = new Job();
                  job.setref(container);
                  job.start(); } } }
```

Theorem The value of the `counter` never decreases.

This has to be formulated more carefully to account for Java's 32-bit `int` arithmetic.

We Have Proved Progress Properties

The theorem above is a Safety property.

We have also proved a Progress property:

The value of the `counter` will increase.

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.