Machines Reasoning About Machines

Part 2

How to Model Machines and

Prove Theorems about Code

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
M1

An M1 state consists of:

- program counter (pc)
- local variables (locals)
- push down stack (stack)
- program to run (program)
ICONST 23 \leftarrow pc
ILOAD 1
IADD
ISTORE 1

\cdots

0 \quad [17 \quad 12]
\quad pc \quad locals \quad stack \quad program
ICONST 23
ILOAD 1
IADD \( \leftarrow pc \)
ISTORE 1

\[
\begin{array}{cccc}
2 & [17 & 12] & 23 \\
pc & locals & stack & program
\end{array}
\]

...
ICONST 23
ILOAD 1
IADD
ISTORE 1 \( \leftarrow pc \)

\[
\begin{array}{cccc}
3 & [17 \ 12] & 35 & \ldots \\
p_{c} & locals & stack & program \\
\end{array}
\]
ICONST 23
ILOAD 1
IADD
ISTORE 1

... ← pc

4 [17 35]

pc  locals  stack  program
If \texttt{locals[1]} is the variable \texttt{a}, then this is the compiled code for 
\texttt{``a := 23+a;''}
Consider g

(defun g (n a)
  (if (zp n)
      a
      (g (- n 1) (* n a))))
The M1 Program

local 0 1
var name n a

(defconst *g*
  '((ICONST 1)
    (ISTORE 1) ; a := 1
    ...)))
local   0  1
var name   n   a

; loop
   (ILOAD 0)
   (IFEQ 10)  ; if n=0 goto end
   (ILOAD 0)
   (ILOAD 1)
   (IMUL)
   (ISTORE 1)  ; a := n * a
...

local 0 1

var name n a

(ILOAD 0)
(ICONST 1)
(ISUB)
(ISTORE 0) ; n := n-1
(GOTO -10) ; goto loop

; end

(ILOAD 1)
(HALT)))); ; ‘‘return’’ a
The Plan

Formalize M1 states and other basic utilities

Formalize the semantics of each instruction

Formalize the “fetch-execute” cycle
Formalizing M1

(defun make-state (pc locals stack program)
  (cons pc
    (cons locals
      (cons stack
        (cons program nil))))))
Formalizing M1

(defun make-state (pc locals stack program)
  (list pc locals stack program))
Formalizing M1

(defun make-state (pc locals stack program)  
  (list pc locals stack program))

(defun pc (s) (nth 0 s))
(defun locals (s) (nth 1 s))
(defun stack (s) (nth 2 s))
(defun program (s) (nth 3 s))
(defun opcode (inst) (nth 0 inst))
(defun arg1 (inst) (nth 1 inst))
(defun arg2 (inst) (nth 2 inst))

(opcode '(ICONST 23)) ⇒ ICONST
(arg1 '(ICONST 23)) ⇒ 23
(defun push (x stk) (cons x stk))
(defun top (stk) (car stk))
(defun pop (stk) (cdr stk))

(push 3 '(2 1)) ⇒ (3 2 1)
(top '(3 2 1)) ⇒ 3
(pop '(3 2 1)) ⇒ (2 1)
Aside We might:

(defthm top-push
  (equal (top (push e stk)) e))

(defthm pop-push
  (equal (pop (push e stk)) stk))

(in-theory (disable top pop push))

to hide the representation of stacks.
(defun do-inst (inst s)
  (if (equal (opcode inst) 'ICONST)
    (execute-ICONST inst s)
    (if (equal (opcode inst) 'ILOAD)
      (execute-ILOAD inst s)
      (if (equal (opcode inst) 'ISTORE)
        (execute-ISTORE inst s)
        (if (equal (opcode inst) 'IADD)
          (execute-IADD inst s)
          ...)))))
(defun execute-ICONST (inst s)
  (make-state (+ 1 (pc s))
    (locals s)
    (push (arg1 inst) (stack s))
    (program s)))
(defun execute-ILOAD (inst s)
  (make-state (+ 1 (pc s))
    (locals s)
    (push (nth (arg1 inst)
              (locals s))
          (stack s))
    (program s)))
(defun execute-ISTORE (inst s)
  (make-state (+ 1 (pc s))
   (update-nth (arg1 inst)
     (top (stack s))
     (locals s))
   (pop (stack s))
   (program s)))
(defun update-nth (n v x)
  (if (zp n)
      (cons v (cdr x))
      (cons (car x)
            (update-nth (- n 1) v (cdr x)))))

(update-nth 1 35 '(17 12)) => (17 35)
(defun execute-IMUL (inst s)
  (make-state (+ 1 (pc s))
    (locals s)
    (push (* (top (pop (stack s)))
          (top (stack s)))
          (pop (pop (stack s))))
    (program s)))
(defun execute-IFEQ (inst s)
  (make-state (if (equal (top (stack s)) 0)
               (+ (arg1 inst) (pc s))
               (+ 1 (pc s)))
  (locals s)
  (pop (stack s))
  (program s)))
(defun do-inst (inst s)
  (if (equal (opcode inst) 'ICONST)
      (execute-ICONST inst s)
    (if (equal (opcode inst) 'ILOAD)
        (execute-ILOAD inst s)
      (if (equal (opcode inst) 'ISTORE)
          (execute-ISTORE inst s)
        (if (equal (opcode inst) 'IADD)
            (execute-IADD inst s)
          ...
      )))
...)
(defun next-inst (s)
   (nth (pc s) (program s)))

(defun step (s)
   (do-inst (next-inst s) s))
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched) (step s)))))

Sched is a “schedule” telling us how many steps to take.

Only its length matters.
Aside

In more sophisticated models, `sched` is a list of “thread identifiers” and tells us which thread to step next.

It may also specify “inputs” that arrive on a chip’s pins at that cycle.

It may also specify “non-deterministic” decisions for that cycle.
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched)
            (step s)))))
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched)
           (run (cadr sched)
                (step (car sched) s))))}

Terminating Computations

When is a state halted?

(defun haltedp (s)
  (equal (step s) s))

How long does it take a program to halt?
((ICONST 1) ; 0
  (ISTORE 1) ; 1   a := 1
  (ILOAD 0) ; 2   loop
  (IFEQ 10) ; 3   if n=0 go end
  (ILOAD 0) ; 4
  (ILOAD 1) ; 5
  (IMUL) ; 6
  (ISTORE 1) ; 7   a := n*a
  (ILOAD 0) ; 8
  (ICONST 1) ; 9
  (ISUB) ; 10
  (ISTORE 0) ; 11  n := n-1
  (GOTO -10) ; 12  go loop
  (ILOAD 1) ; 13  end
  (HALT)))) ; 14  ‘‘return’’ a
(ICONST 1) ; 0
(ISTORE 1) ; 1 a := 1
(ILOAD 0) ; 2 loop
(IFEQ 10) ; 3 if n=0 go end
(ILOAD 0) ; 4
(ILOAD 1) ; 5
(IMUL) ; 6
(ISTORE 1) ; 7 a := n*a
(ILOAD 0) ; 8
(ICONST 1) ; 9
(ISUB) ; 10
(ISTORE 0) ; 11 n := n-1
(GOTO -10) ; 12 go loop
(ILOAD 1) ; 13 end
(HALT))) ; 14 ‘‘return’’ a
((ICONST 1) ; 0
  (ISTORE 1) ; 1  a := 1
  (ILOAD 0) ; 2  loop
  (IFEQ 10) ; 3  if n=0 go end
  (ILOAD 0) ; 4
  (ILOAD 1) ; 5
  (IMUL) ; 6
  (ISTORE 1) ; 7  a := n*a
  (ILOAD 0) ; 8
  (ICONST 1) ; 9
  (ISUB) ; 10
  (ISTORE 0) ; 11  n := n-1
  (GOTO -10) ; 12  go loop
  (ILOAD 1) ; 13  end
  (HALT))) ; 14  ‘‘return’’ a
'((ICONST 1) ; 0
  (ISTORE 1) ; 1 a := 1
  (ILOAD 0) ; 2 loop
  (IFEQ 10) ; 3 if n=0 go end
  (ILOAD 0) ; 4
  (ILOAD 1) ; 5
  (IMUL) ; 6
  (ISTORE 1) ; 7 a := n*a
  (ILOAD 0) ; 8
  (ICONST 1) ; 9
  (ISUB) ; 10
  (ISTORE 0) ; 11 n := n-1
  (GOTO -10) ; 12 go loop
  (ILOAD 1) ; 13 end
  (HALT)))); ; 14 ‘‘return’’ a
A Schedule for \texttt{g}

(defun g-sched (n)
  (ap (repeat 'TICK 2)
       (g-loop-sched n)))

(defun g-loop-sched (n)
  (if (zp n)
      (repeat 'TICK 3)
      (ap (repeat 'TICK 11)
           (g-loop-sched (- n 1)))))

Running \texttt{g}

\begin{verbatim}
(defun test-g (n)
  (top
   (stack
    (run (g-sched n)
      (make-state 0 (list n 0) nil *g*))))
  (test-g 5) ⇒ 120
\end{verbatim}
Demo 1
M1 inherits a lot of power from ACL2.

We’re executing about 360,000 instructions/sec on this laptop.

But how does M1 compare to the JVM?
Sun JVM Specification

ILOAD
Operation
Load int from local variable
Format (2 bytes)
ILOAD index
Form
21 (0x15)
Operand Stack
... ⇒ ..., value
Description

The \textit{index} is an unsigned byte that must be an index into the local variable array of the current frame. The local variable at \textit{index} must contain an \texttt{int}. The value of the local variable at \textit{index} is pushed onto the operand stack.
ILOAD
Operation
Load int from local variable
Format (2 bytes)
ILOAD index
Form
21 (0x15)
Operand Stack
... \Rightarrow ... , value
ILOAD \textit{typed!}

Operation
Load int from local variable

Format (2 bytes)
ILOAD \textit{index}

Form
21 (0×15)

Operand Stack
... ⇒ ..., value
ILOAD Operation

32-bit arithmetic!

Load int from local variable

Format (2 bytes)

ILOAD index

Form

21 (0x15)

Operand Stack

... ⇒ ..., value
ILOAD Operation
   Load int from local variable

Format (2 bytes) instruction stream
   ILOAD index is unparsed bytes

Form
   21 (0x15)

Operand Stack
   ... ⇒ ..., value
Description *threads and method calls*

The *index* is an unsigned byte that must be an index into the local variable array of the *current frame*. The local variable at *index* must contain an *int*. The value of the local variable at *index* is pushed onto the operand stack.
Comparison with the JVM

• specification style is very similar

• functionality is similar but the JVM is much richer

• M1 is missing procedure call (activation stack), Objects (heap), threads (thread table and monitors), and Exceptions

• M1 is missing class loading and verification
It is possible to “grow” M1 into a complete JVM. *But we don’t have time to deal with them here!*
A High Level Language

It is easy to write a compiler from a simple language of `while` and assignments to M1 code.

We could verify that the compiler is correct.

*But we don’t have time to explore these issues here!*
First Conclusion

Three advantages of operational semantics:

- relatively easy to formalize idiosyncratic custom languages
- easy to relate to implementation or an informal specification
- formal models are executable (highly desired by "customers")
Our Next Goal

Learn how to prove M1 code correct.
Fact 1

Given an operational semantics, *symbolic execution* of code is just substitution-of-equals-for-equals, i.e., rewriting.
'((ICONST 1) ; 0
  (ISTORE 1) ; 1  a := 1
  (ILOAD 0) ; 2  loop
  (IFLE 10) ; 3  if n=0 go end
  (ILOAD 0) ; 4
  (ILOAD 1) ; 5
  (IMUL) ; 6
  (ISTORE 1) ; 7  a := n*a
  (ILOAD 0) ; 8
  (ICONST 1) ; 9
  (ISUB) ; 10
  (ISTORE 0) ; 11  n := n-1
  (GOTO -10) ; 12  go loop
  (ILOAD 1) ; 13  end
  (HALT))) ; 14  ‘‘return’’ a
((ICONST 1) ; 0
 (ISTORE 1) ; 1  a := 1
 (ILOAD 0) ; 2 loop
 (IFLE 10) ; 3 if n=0 go end
 (ILOAD 0) ; 4  pc
 (ILOAD 1) ; 5
 (IMUL) ; 6
 (ISTORE 1) ; 7  a := n*a
 (ILOAD 0) ; 8
 (ICONST 1) ; 9
 (ISUB) ; 10
 (ISTORE 0) ; 11  n := n-1
 (GOTO -10) ; 12 go loop
 (ILOAD 1) ; 13 end
 (HALT))) ; 14 ‘‘return’’ a
(run (repeat 'TICK 9)
  (make-state 4 ; \(\leftarrow pc\)
    (list n a)
    stk
    *g*))}
=  
(run (repeat 'TICK 8)  
    (step (make-state 4 
        (list n a) 
        stk 
        *g*)))))
(run (repeat 'TICK 8)
  (step (make-state 4 ; ⇒ (ILOAD 0)
          (list n a)
          stk
          *g*)))
(run (repeat 'TICK 8)
 (execute-LOAD '(ILOAD 0)
  (make-state 4
   (list n a)
   stk
   *g*))))
(run (repeat 'TICK 8)
  (make-state 5
    (list n a)
    (push n stk)
    *g*))
}
(run (repeat 'TICK 8)
    (make-state 5  ; ⇒ (ILOAD 1)
        (list n a)
        (push n stk)
        *g*)))
(run (repeat 'TICK 7)
  (make-state 6 ; ⇒ (IMUL)
    (list n a)
    (push a (push n stk))
    *g*)))
(run (repeat 'TICK 6)
  (make-state 7 ; ⇒ (ISTORE 1)
    (list n a)
    (push (* n a) stk)
    *g*))
(run (repeat ’TICK 5)
  (make-state 8 ; ⇒ (ILOAD 0)
   (list n (* n a))
   stk
   *g*)))
(run (repeat 'TICK 4)
   (make-state 9  ; ⇒ (ICONST 1)
       (list n (* n a))
       (push n stk)
       *g*)))
(run (repeat 'TICK 3)
  (make-state 10 ; ⇒ (ISUB)
    (list n (* n a))
    (push 1 (push n stk))
    *g*))
(run (repeat 'TICK 2)
  (make-state 11 ; ⇒ (ISTORE 0)
    (list n (* n a))
    (push (- n 1) stk)
    *g*)))
==

(run (repeat ’TICK 1)
  (make-state 12 ; ⇒ (GOTO -10)
    (list (- n 1) (* n a))
    stk
    *g*))
(run (repeat 'TICK 0)
  (make-state 2
    (list (- n 1) (* n a))
    stk
    *g*))
≡

(run nil
  (make-state 2
    (list (- n 1) (* n a))
    stk
    *g*)))
(make-state 2
   (list (- n 1) (* n a))
   stk
   *g*)
Demo 2

Theorem?
Provided n and a are natural numbers,
(equal (run (repeat 'TICK 9)
  (make-state 4
    (list n a)
    stk
    *g*)))

???)
Fact 2

Concatenation (ap) (of schedules) is just *sequential composition*.

**Theorem.** run-ap

(equal (run (ap a b) s)
  (run b (run a s))))
Demo 3
Aside

Note that a virtue of having an operational semantics expressed in a formal logic is that we can prove theorems about the semantics, independent of any particular program.
Having proved \textit{run-ap}, whenever the system encounters:

\[\text{(run (ap } \alpha \ \beta) \ \sigma)\]

it will replace it by

\[\text{(run } \beta \text{ (run } \alpha \ \sigma))\]

i.e., decompose theorems with complicated schedules (created by \textit{ap}) into symbolic runs of the pieces.
To Prove Code Correct

Proceed in two steps:

• Step 1: code implements algorithm – innermost loop first

• Step 2: algorithm implements specification

Step 2 doesn’t involve code or the operational semantics.
Demo 4
Aside

We have completely characterized the effects of executing *g*.

- it is possible to prove partial correctness ("if *g* is halted then ...")

- it is possible to prove correctness without characterizing every effect
in security contexts, knowing every effect is often very important
Corollaries

Provided $n$ is a natural number,

- $*g*$ “returns” (! $n$)

- $*g*$ halts after (len (g-sched $n$)) steps

It is also possible to prove that $*g*$ never halts if $n < 0$. 
Second Conclusion

It is possible to reason mechanically about code under an operational semantics.

It is relatively easy to build a verification system for a custom language using a general-purpose theorem prover.

With care, this approach can be scaled to more complex models. Recall M6.
Next Time

How to drive ACL2.