Mechanized Operational Semantics

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

Marktoberdorf Summer School 2008

(Lecture 2: An Operational Semantics)

Java Virtual Machine

We have a precise mathematical model of the Java Virtual Machine, called M6 (*Model 6*)

It is too complicated to present here (160 pages).

We will look at a simpler model, M1 (3 pages).

M1

An M1 *state* consists of:

- program counter (pc)
- local variables (locals)
- push down stack (stack)
- program to run (program)

`PUSH 23` $\Leftarrow$ pc

`LOAD 1`

`ADD`

`STORE 1`

`...`

0	[17 12]		
pc	*locals*	*stack*	*program*

```
PUSH 23
LOAD 1      ⇐ pc
ADD
STORE 1
...
```

`1`	`[17 12]`	`23`	
pc	*locals*	*stack*	*program*

```
PUSH 23
LOAD 1
ADD        ⇐ pc
STORE 1
...
```

12

2 [17 12] 23

pc *locals* *stack* *program*

```
PUSH 23
LOAD 1
ADD
STORE 1    ⇐ pc
...
```

3	[17 12]	35	
pc	*locals*	*stack*	*program*

```
PUSH 23
LOAD 1
ADD
STORE 1
. . .       ⇐ pc
```

4	[17 35]		
pc	*locals*	*stack*	*program*

```
PUSH 23
LOAD 1
ADD
STORE 1
...
```

4	`[17 35]`		
pc	*locals*	*stack*	*program*

If *locals[1]* is the variable a, then this is the compiled code for “a = 23+a;”

Recall g

```
(defun g (n a)
  (if (zp n)
      a
    (g (- n 1) (* n a))))
```

The M1 Program

We use *locals[0]* to hold `n` and *locals[1]* to hold `a`.

```
(defconst *g*
  '((PUSH 1)
    (STORE 1)    ; a := 1
    ...))
```

```
; loop
    (LOAD 0)
    (IFLE 10)    ; if n<=0 go end
    (LOAD 0)
    (LOAD 1)
    (MUL)
    (STORE 1)    ; a := n*a
    ...
    (LOAD 0)
    (PUSH 1)
    (SUB)
    (STORE 0)    ; n := n-1
    (GOTO -10)   ; go loop
; end
    (LOAD 1)
    (RETURN)))
```

M1 versus JVM

```
% cat Fact.java
% javac Fact.java
% javap -c Fact
```

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) (car inst))
(defun arg1   (inst) (nth 1 inst))
(defun arg2   (inst) (nth 2 inst))

(opcode '(PUSH 23)) ⇒ PUSH
(arg1 '(PUSH 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)
```

```
(defun do-inst (inst s)
  (if (equal (opcode inst) ’PUSH)
      (execute-PUSH  inst s)
    (if (equal (opcode inst) ’LOAD)
        (execute-LOAD  inst s)
      (if (equal (opcode inst) ’STORE)
          (execute-STORE  inst s)
        (if (equal (opcode inst) ’ADD)
            (execute-ADD   inst s)
            ...
```

```
(defun execute-PUSH (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (arg1 inst) (stack s))
              (program s)))
```

```
(defun execute-LOAD (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (nth (arg1 inst)
                         (locals s))
                    (stack s))
              (program s)))
```

```
(defun execute-STORE (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-MUL (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-GOTO (inst s)
  (make-state (+ (arg1 inst) (pc s))
              (locals s)
              (stack s)
              (program s)))
```

```
(defun execute-IFLE (inst s)
  (make-state (if (<= (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) 'PUSH)
      (execute-PUSH  inst s)
    (if (equal (opcode inst) 'LOAD)
        (execute-LOAD  inst s)
      (if (equal (opcode inst) 'STORE)
          (execute-STORE  inst s)
        (if (equal (opcode inst) 'ADD)
            (execute-ADD   inst s)
            ...
```

Aside: HOL

If we had a higher order logic:

- instruction: state $\longrightarrow$ state
- `do-inst`: *apply*

```
(defun do-inst (inst s)
  (if (equal (opcode inst) 'PUSH)
      (execute-PUSH  inst s)
    (if (equal (opcode inst) 'LOAD)
        (execute-LOAD  inst s)
      (if (equal (opcode inst) 'STORE)
          (execute-STORE  inst s)
        (if (equal (opcode inst) 'ADD)
            (execute-ADD   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.

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

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

Terminating Computations

When is a state halted?

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

Recall Program g

```
(defconst *g*
  '((PUSH 1)     ;  0
    (STORE 1)    ;  1   a := 1
    (LOAD 0)     ;  2 loop
    (IFLE 10)    ;  3   if n<=0 go end
    (LOAD 0)     ;  4
    (LOAD 1)     ;  5
    (MUL)        ;  6
    (STORE 1)    ;  7   a := n*a
    (LOAD 0)     ;  8
    ...))
```

How long does it take to run `g`?

Let's construct a schedule for `g`.

More precisely, let's write a function that takes `g`'s input `n` and returns a schedule to run `g` on `n`.

```
'((PUSH 1)     ;  0
  (STORE 1)    ;  1   a := 1
  (LOAD 0)     ;  2 loop
  (IFLE 10)    ;  3   if n<=0 go end
  (LOAD 0)     ;  4
  (LOAD 1)     ;  5
  (MUL)        ;  6
  (STORE 1)    ;  7   a := n*a
  (LOAD 0)     ;  8
  (PUSH 1)     ;  9
  (SUB)        ; 10
  (STORE 0)    ; 11   n := n-1
  (GOTO -10)   ; 12   go loop
  (LOAD 1)     ; 13 end
  (RETURN)))   ; 14   return a
```

```
'((PUSH 1)      ;  0
  (STORE 1)     ;  1    a := 1
  (LOAD 0)      ;  2 loop
  (IFLE 10)     ;  3    if n<=0 go end
  (LOAD 0)      ;  4
  (LOAD 1)      ;  5
  (MUL)         ;  6
  (STORE 1)     ;  7    a := n*a
  (LOAD 0)      ;  8
  (PUSH 1)      ;  9
  (SUB)         ; 10
  (STORE 0)     ; 11    n := n-1
  (GOTO -10)    ; 12    go loop
  (LOAD 1)      ; 13 end
  (RETURN))     ; 14    return a
```

```
'((PUSH 1)      ;  0
  (STORE 1)     ;  1    a := 1
  (LOAD 0)      ;  2 loop
  (IFLE 10)     ;  3    if n<=0 go end
  (LOAD 0)      ;  4
  (LOAD 1)      ;  5
  (MUL)         ;  6
  (STORE 1)     ;  7    a := n*a
  (LOAD 0)      ;  8
  (PUSH 1)      ;  9
  (SUB)         ; 10
  (STORE 0)     ; 11    n := n-1
  (GOTO -10)    ; 12    go loop
  (LOAD 1)      ; 13 end
  (RETURN))     ; 14    return a
```

```
'((PUSH 1)      ;  0
  (STORE 1)     ;  1    a := 1
  (LOAD 0)      ;  2 loop
  (IFLE 10)     ;  3    if n<=0 go end
  (LOAD 0)      ;  4
  (LOAD 1)      ;  5
  (MUL)         ;  6
  (STORE 1)     ;  7    a := n*a
  (LOAD 0)      ;  8
  (PUSH 1)      ;  9
  (SUB)         ; 10
  (STORE 0)     ; 11    n := n-1
  (GOTO -10)    ; 12    go loop
  (LOAD 1)      ; 13 end
  (RETURN))     ; 14    return a
```

A Schedule for `g`

```
(defun g-sched (n)
  (append (repeat 0 2)
          (g-sched-loop n)))

(defun g-sched-loop (n)
  (if (zp n)
      (repeat 0 4)
    (append (repeat 0 11)
            (g-sched-loop (- n 1)))))
```

Running `g`

```
(defun run-g (n)
  (top
   (stack
    (run (g-sched n)
         (make-state 0 (list n 0) nil *g*)))))
```

`(run-g 5)` ⇒ `120`

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?

ILOAD

Operation

Load `int` from local variable

Format (2 bytes)

ILOAD *index*

Form

21 (0x15)

Operand Stack

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

Description

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.

ILOAD

Operation

Load `int` from local variable

Format (2 bytes)

ILOAD *index*

Form

21 (0x15)

Operand Stack

... ⇒ ..., value

ILOAD *typed!*

Operation

Load `int` from local variable

Format (2 bytes)

ILOAD *index*

Form

21 (0x15)

Operand Stack

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

ILOAD

Operation *32-bit arithmetic!*

Load `int` from local variable

Format (2 bytes)

ILOAD *index*

Form

21 (0x15)

Operand Stack

... $\Rightarrow$ ..., 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

It is possible to “grow” M1 into a complete JVM.

A High Level Language

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

Demo 2

To see the implementation of the compiler, read the preliminary material prepared for this Summer School.

Conclusion

Two advantages of operational semantics:

- easy to relate to implementation or an informal specification
- executable

ACL2 “customers” *really like* the ability to run their models.

Next Time

But can we prove anything about a model like this?