Proving Theorems about Java-like Byte Code

J Strother Moore*

May 24, 1999

Abstract

We describe a formalization of an abstract machine very similar to the Java Virtual Machine but far simpler. We develop techniques for specifying the properties of classes and methods for this machine. We develop techniques for mechanically proving theorems about classes and methods. We discuss two such proofs, that of a static method implementing the factorial function and of an instance method that destructively manipulates objects in a way that takes advantage of inheritance. We conclude with a brief discussion of the advantages and disadvantages of this approach. The formalization and proofs are done with the ACL2 theorem proving system.

1 Specification of the TJVM

The Java Virtual Machine (JVM) [10] is a stack-based, object-oriented, type-safe byte-code interpreter on which compiled Java programs are executed.

We develop a simplified JVM for the purpose of exploring verification issues related to proofs about object-oriented byte code. We refer to our machine as a "toy JVM" or "TJVM." Because we are interested in formal, mechanically checked proofs, we formalize the TJVM in a formal, mechanized logic, namely ACL2: A Computational Logic for Applicative Common Lisp. The tradition of formalizing machines in ACL2, and its predecessor, Boyer and Moore's Nqthm, is well established [1, 14, 11, 3, 5] and we follow in those well-trodden footsteps. Indeed, our TJVM is just a simplification of Rich Cohen's "defensive JVM," [6], which was formalized at Computational Logic, Inc., in the standard ACL2/Nqthm style. That style employs an operational semantics, in which the state of the machine is represented as a Lisp object. An interpreter for the machine's programming language is defined as a Lisp function. The main purpose of this paper is to illustrate how those established techniques can be applied in an object-oriented setting.

This paper provides a brief sketch of our TJVM. The details may be obtained at http://www.cs.utexas.edu/users/moore/publications/tjvm/index.html.

The state of the TJVM is a triple consisting of a *call stack* of "frames," a *heap*, and a *class table*. A *frame* contains four fields, a *program counter*, a variable binding environment called the *locals* of the frame, an operand *stack*, and the byte code *program* of the method being evaluated. The *heap* is an association of integer *addresses* to "objects," which are "instances" of classes. An *instance* is a list of n tables, one for each of the n superclasses of the object. Each table enumerates the fields of a given class and specifies the contents of those fields in this particular instance. Finally, the *class table* is a list of *class declarations*, each of which specifies a class name, the names of its superclasses, the names of its fields, and the "method declarations" of the class. A *method declaration* specifies a method name, its formal parameters, and the byte coded body of the method.

Readers familiar with the JVM will recognize the TJVM as a similar machine. Here is a Java program for computing the factorial function.

*Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

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

Here is an example of a TJVM method declaration corresponding to the compiled code for `fact` above. The comments (on the right, following the semi-colons) indicate the TJVM program counter of the instruction and the JVM byte code produced by Sun's Java compiler.

Fact:

```
("fact" (n)
        (load n)                        ;  0       iload_0
        (ifle 8)                        ;  1       ifle 12
        (load n)                        ;  2       iload_0
        (load n)                        ;  3       iload_0
        (push 1)                        ;  4       iconst_1
        (sub)                           ;  5       isub
        (invokestatic "Math" "fact" 1) ;  6       invokestatic <Method int fact(int)>
        (mul)                           ;  7       imul
        (xreturn)                       ;  8       ireturn
        (push 1)                        ;  9       iconst_1
        (xreturn))                      ; 10       ireturn
```

This method declaration, **Fact**, might be found in the class declaration of the `"Math"` class on the TJVM. The name of the method is `"fact"`. It has one formal parameter, `n`, and a byte code program of eleven instructions. **Fact** is actually a Lisp constant whose first element is the string `"fact"`, whose second element is the list containing the single symbol `n`, etc.

We discuss later the correspondence between TJVM byte codes and JVM byte codes, but a shallow correspondence is obvious. On the TJVM, methods refer to their local variables by name; on the JVM methods refer to their locals by position. The "`8`" in the TJVM `ifle` instruction is an instruction offset by which the program counter is incremented. The corresponding offset on the JVM counts bytes rather than instructions and some JVM instructions take more than one byte. Finally, the JVM has typed instructions, e.g., the JVM's `iload` loads an integer-valued variable on the stack while the TJVM's `load` loads any value.

When this program is invoked on the TJVM, one actual parameter, n, is popped from the operand stack of the topmost frame of the TJVM call stack; a new frame is built and pushed onto the call stack of the TJVM state. The new frame has program counter 0 and the eleven instructions above as the program. The locals of the new frame bind `n` to the actual n. The operand stack of the new frame is empty.

The program operates as follows. The parenthesized numbers refer to program counter values. (0) The local value, n, of `n` is pushed onto the operand stack. (1) The `ifle` instruction pops the operand stack and compares the item obtained, here n, to 0. If $n \leq 0$, the program counter is incremented by `8`; otherwise it is incremented by `1`. (9-10) In the former case, the program pushes a `1` on the operand stack and returns one result to the caller. The JVM uses a special, single byte instruction for pushing the constant `1`, while the TJVM has one general-purpose `push` instruction for pushing any constant. (2) In the case that $n > 0$, the program pushes n, (3-5) pushes $n-1$ (by (3) pushing n and (4) 1 and (5) executing a `sub` which pops two items off the operand stack and pushes their difference), (6) invokes this procedure recursively on one actual, in this case, the $n-1$ on the stack, obtaining one result, here $(n-1)!$, which is pushed on the stack in place of the actual, (7) multiplies the top two elements of the stack, and (8) returns that one result to the caller.

From the above description it should be clear how we define the semantics of the instructions illustrated above. For example, here is the function which gives semantics to the `add` instruction, `(add)`.

```
(defun execute-ADD (inst s)
  (declare (ignore inst))
  (make-state
   (push (make-frame (+ 1 (pc (top-frame s)))
                     (locals (top-frame s))
                     (push (+ (top (pop (stack (top-frame s))))
                              (top (stack (top-frame s))))
                           (pop (pop (stack (top-frame s)))))
                     (program (top-frame s)))
         (pop (call-stack s)))
   (heap s)
   (class-table s)))
```

This Lisp function – which in the ACL2 setting is taken as an *axiom* defining the expression **(execute-ADD** *inst s***)** – takes two arguments, the **add** instruction to be interpreted and the TJVM state *s*. Because the TJVM **add** instruction has no operands, **execute-ADD** does not actually need the instruction and so *inst* above is ignored. The function computes the state obtained from *s* by executing an **add**. The new state contains a modified call stack but the same heap and class table as *s*. The call stack is modified by changing only its topmost frame so that the program counter is incremented by one and the operand stack is modified by popping two items off of it and pushing their sum. The locals and program of the frame are unchanged. We call **execute-ADD** the *semantic function* for the TJVM **add** instruction.

Each instruction on the TJVM is defined by an analogous semantic function. The semantic function for the instruction **(new "***class***")**, which constructs a new instance of the class named "*class*", is as follows.

```
(defun execute-NEW (inst s)
  (let* ((class (arg1 inst))
         (table (class-table s))
         (obj (build-an-instance
                     (cons class
                           (class-decl-superclasses
                            (bound? class table)))
                     table))
         (addr (length (heap s))))
    (make-state
     (push (make-frame (+ 1 (pc (top-frame s)))
                       (locals (top-frame s))
                       (push (list 'REF addr)
                             (stack (top-frame s)))
                       (program (top-frame s)))
           (pop (call-stack s)))
     (bind addr obj (heap s))
     (class-table s))))
```

Informally, the function builds an uninitialized instance, *obj*, of the given *class* obtaining the class' fields and superclasses from the class table of the state. It also obtains a new address, *addr*, in the heap (namely, the length of the current heap). It then modifies the top frame of the call stack by incrementing the program counter and pushing onto the operand stack a reference to the new address, namely the list object **(REF** *addr***)**. It modifies the heap by binding *addr* to the instance, *obj*.

The semantic function for the instruction **(invokevirtual "***class***" "***name***"** *n***)** is shown below. Informally, the instruction obtains a reference, *ref*, to the "this" object of the invocation by looking down the operand stack an appropriate distance, given *n*, the number of formals of the named method. It then obtains the class, *class*, of the referenced object and uses the given method *name* and object *class* to determine the nearest appropriate method, *method*. Let *vars* be the formal parameters of the resolved *method*, extended with one additional formal, named **this**, and let *prog* be the byte code program for *method*.

```
(defun execute-INVOKEVIRTUAL (inst s)
  (let* ((name (arg2 inst))
         (n (arg3 inst))
         (ref (top (popn n (stack (top-frame s)))))
         (class (class-name-of-ref ref (heap s)))
         (method
          (lookup-method name
                         class
                         (class-table s)))
         (vars (cons 'this (method-formals method)))
         (prog (method-program method)))
    (make-state
     (push (make-frame 0
                       (reverse
                        (bind-formals (reverse vars)
                                      (stack (top-frame s))))
                       nil
                       prog)
           (push (make-frame (+ 1 (pc (top-frame s)))
                             (locals (top-frame s))
                             (popn (length vars)
                                   (stack (top-frame s)))
                             (program (top-frame s)))
                 (pop (call-stack s))))
     (heap s)
     (class-table s))))
```

The instruction then modifies the existing top frame of the call stack by incrementing the program counter and popping $n + 1$ items off the operand stack. It then pushes a new frame poised to execute the resolved method, initializing the locals in the new frame by binding *vars* to the items just popped.

After defining a semantic function for each TJVM instruction, we define:

```
(defun do-inst (inst s)
  (case (op-code inst)
    (PUSH           (execute-PUSH inst s))
    (POP            (execute-POP  inst s))
    (LOAD           (execute-LOAD inst s))
    (STORE          (execute-STORE inst s))
    (ADD            (execute-ADD inst s))
    (SUB            (execute-SUB inst s))
    (MUL            (execute-MUL inst s))
    (GOTO           (execute-GOTO inst s))
    (IFEQ           (execute-IFEQ inst s))
    (IFGT           (execute-IFGT inst s))
    (INVOKEVIRTUAL  (execute-INVOKEVIRTUAL inst s))
    (INVOKESTATIC   (execute-INVOKESTATIC inst s))
    (RETURN         (execute-RETURN inst s))
    (XRETURN        (execute-XRETURN inst s))
    (NEW            (execute-NEW inst s))
    (GETFIELD       (execute-GETFIELD inst s))
    (PUTFIELD       (execute-PUTFIELD inst s))
    (HALT           s)
    (otherwise s)))
```

so that `(do-inst` *inst s*`)` returns the state obtained by executing *inst* in state *s*. The definition enumerates the instructions supported on the TJVM and calls the appropriate semantic function.

Each of the supported instructions is modeled after one or more JVM instructions. TJVM instructions are generally simpler than their JVM counterparts. For example, we are not concerned with access attributes,

types, resource limitations, or exceptions on the TJVM. The TJVM classes provide for "instance fields" but do not provide the JVM's "static fields." Unlike their counterparts on the JVM, our `INVOKEVIRTUAL` and `INVOKESTATIC` do not take "signatures." On the JVM, a method's "this" object is in local variable 0; on the TJVM it is an implicit formal parameter named `this`. Like its JVM counterpart, the method actually invoked by our `INVOKEVIRTUAL` depends on the "this" object: both invoke the nearest method of the given name found in the superclass chain of the object, but the JVM discriminates between candidate methods via their signatures and we do not. That is, the JVM supports "overloading" and the TJVM does not.

The "single stepper" for the TJVM is

```
(defun step (s)
  (do-inst (next-inst s) s))
```

where `next-inst` retrieves the instruction indicated by the program counter in the topmost frame of the call stack.

The TJVM is then defined as an iterated step function:

```
(defun tjvm (s n)
  (if (zp n)
      s
    (tjvm (step s) (- n 1))))
```

Thus `(tjvm s n)` is the result of applying `step` to s n times, or `(step`^{n} s`)`.

The TJVM can be viewed as a simplification of the JVM. The TJVM is in fact a simplification of Rich Cohen's "defensive JVM", [6], which includes many more JVM instructions and deals carefully with the preconditions assumed for each instruction. In principle, Cohen's specification could be used to analyze whether a given byte code verifier is sufficient to guarantee the absence of certain classes of runtime errors. Both the TJVM and the defensive JVM omit major aspects of the JVM, including floating point numbers, arrays, multiple threads, exceptions, and native methods. All but native methods could be formalized in an implementation independent way, following the basic approach.

2 Example TJVM Executions

Having defined the TJVM in Lisp, it is possible to execute it on concrete data. Consider the byte code for the `"fact"` program shown earlier in the constant **Fact**. Let **Math-class** denote the list constant partially displayed below.

Math-class:
`("Math" ("Object") () (...` **Fact** `...))`.

This is a class declaration for a class called `"Math"` which is an extension of the class named `"Object"` (literally, the declaration says that the superclass chain of `"Math"` is the list containing only the class named `"Object"`). The `"Math"` class contains no fields and its methods are those listed and include **Fact**.

Consider the TJVM state

s_0:
```
(make-state
     (push (make-frame 0
                       nil
                       nil
                       '((push 5)
                         (invokestatic "Math" "fact" 1)
                         (halt)))
           nil)
     nil
     '( Math-class))
```

This state is poised to execute the three instruction program above, starting with the `(push 5)` at program counter 0. Note that the program pushes 5 on the stack and then invokes `"fact"`. The class table for the state includes our **Math-class**, so `"fact"`, here, means the byte code given in **Fact**.

The Lisp expression `(top (stack (top-frame (tjvm` $\mathbf{s}_0$ `52))))` steps $\mathbf{s}_0$ 52 times, and then recovers the topmost item on the operand stack of the topmost frame of the call stack. Evaluating this expression produces 120, which is indeed 5!.

How did we know to take 52 steps? The answer is: by analysis of the code in **Fact**. The following function, which we call the "clock function" for `"fact"`, returns the number of TJVM instructions required to execute `(invokestatic "Math" "fact" 1)` on n. The function was written based on an inspection of the byte code in **Fact**.

```
(defun fact-clock (n)
  (if (zp n)
      5
    (++ 7
        (fact-clock (- n 1))
        2)))
```

Here, `++` is just the normal arithmetic addition function. `Fact-clock` has been written this way (rather than $5 + 9n$) to make it obvious how such functions are generated. To execute a call of `"fact"` on 5 evidently takes 50 TJVM cycles (including the `call`). Thus, the program in $\mathbf{s}_0$ takes 52 cycles.

3 Proofs about TJVM Programs

Of more interest than mere execution is the following theorem we can prove about `"fact"`.

> **Theorem.** `"fact"` is correct:
> Suppose s_0 is a TJVM state whose next instruction is `(invokestatic "Math" "fact" 1)`, where the meaning of the name `"Math"` in the class table is our **Math-class**. Let n be the top of the operand stack in the topmost frame of the call stack of s_0 and suppose n is a natural number. Then the TJVM state obtained by stepping s_0 `(fact-clock` n`)` times is state s_0 with the program counter incremented by one and the n on the operand stack replaced by $n!$. The heap is unchanged.

This informal statement can be phrased formally as follows.

Theorem. `"fact"` is correct:

```
(implies (and (equal (next-inst s0)
                     '(invokestatic "Math" "fact" 1))
              (equal (assoc-equal "Math" (class-table s0))
                     Math-class)
              (equal n (top (stack (top-frame s0))))
              (natp n))
         (equal
          (tjvm s0 (fact-clock n))
          (make-state
           (push (make-frame (+ 1 (pc (top-frame s0)))
                             (locals (top-frame s0))
                             (push (fact n)
                                   (pop (stack (top-frame s0))))
                             (program (top-frame s0)))
                 (pop (call-stack s0)))
           (heap s0)
           (class-table s0)))))
```

The theorem states the *total correctness* of the `"fact"` byte code. Weaker theorems can be stated and proved, but we here focus on theorems of this kind because they are easiest to prove.

We proved this theorem in a very straightforward manner using ACL2. The proof takes 0.33 seconds on a 200 MHz Sun Ultra 2. The theorem only looks complicated because the notation is unfamiliar!

ACL2 is an automatic theorem prover in the sense that its behavior on any given proof attempt is determined by its state immediately prior to the attempt, together with goal-specific hints provided by the user. Of great importance is the set of lemmas the system has already proved. Those lemmas determine how ACL2 simplifies expressions. To configure ACL2 to prove theorems about TJVM we followed the example described in [3]. Roughly speaking, we did the following:

- We proved half a dozen simple arithmetic lemmas; we could have loaded any of several standard ACL2 arithmetic "books."

- We proved lemmas that let ACL2 manipulate the data structures used on the TJVM, including stacks, frames, and states, as abstract data structures. One such theorem is `(top (push` x *stack*`))` $= x$. We then "disabled" the definitions of the primitive stack, frame, and state functions so that their "implementations" in terms of conses were not visible.

- We proved the standard theorem for expanding the single step function, `step`, when the `next-inst` is explicit. We then disabled the step function. This prevents case explosion on what the next instruction is.

- We defined the standard "clock addition" function, `++`, which is really just natural number addition, and disabled it. This allows us to define clock function in the structured style illustrated `fact-clock` and prevents the arithmetic rules from rearranging the expressions and destroying the structural "hints" implicit in the definitions. We proved the rules that allow `tjvm` expressions to be decomposed according to their clock expressions. For example, `(tjvm` s `(++` i j`))` is rewritten to `(tjvm (tjvm` s i`)` j`)`. Thus, when ACL2 encounters, for example, `(tjvm` s_0 `(++ 7 (fact-clock (-` n `1)) 2))` it decomposes it into a run of length seven, followed by a run of length `(fact-clock (-` n `1))`, followed by a run of length two, and each run must be fully simplified to a symbolic state before the next can be simplified (because of the way the step function has been limited).

- Finally we prove the standard "memory management" rules, which in the case of the TJVM tell us the algebra of association lists (used to bind variables, associate programs with method names, method names with methods, fields with their contents, etc.).

Having so configured ACL2, the theorem about `"fact"` above is proved by giving the theorem prover a single hint, namely to do the induction that unwinds the code in `"fact"`.

It is important to realize that the theorem above about `"fact"` contributes to the further configuration of ACL2 in this capacity. The lemma causes the following future behavior of ACL2: Suppose the system encounters an expression of the form `(tjvm` α `(fact-clock` β`))` to simplify. Then it first determines whether the next instruction of the state α is `(invokestatic "Math" "fact" 1)` where the meaning of `"Math"` in α is our **Math-class**, and whether β is on top of the operand stack of α and is a natural number. If so, it replaces `(tjvm` α `(fact-clock` β`))` by the corresponding `make-state` expression in which the program counter has been incremented by one and β has been replaced by `(fact` β`)`.

Thus, after this lemma about `"fact"` is proved, the theorem prover no longer looks at the code for `"fact"`. It steps over `(invokestatic "Math" "fact" 1)` as though it were a primitive instruction that computes the factorial of the top of the stack.

The verification of a system of methods is no harder than the combined verification of each method in the system. This remark, while trivial, has profound consequences if one clearly views the software verification problem as the specification and verification of the component pieces.

4 More Example TJVM Executions

The `"fact"` method does not affect the heap. That is, it does not create any new objects or modify existing objects. How does our verification strategy cope with that? We will consider a simple example of a heap modifying method. But we first illustrate such methods by simple execution. Consider the following TJVM class declaration for a class named `"Point"`, which extends the `"Object"` class and has two fields, named `"x"` and `"y"`. Instances of the `"Point"` class represent points in the Cartesian plane.

Point-class:

```
("Point" ("Object")
         ("x" "y")
         (xIncrement
          inBox))
```

Notice that the class has two methods. The first is defined by the list constant:

xIncrement:

```
("xIncrement" (dx)
              (load this)              ; 0
              (load this)              ; 1
              (getfield "Point" "x")   ; 2
              (load dx)                ; 3
              (add)                    ; 4
              (putfield "Point" "x")   ; 5
              (return))                ; 6
```

We discuss this method now and will display the second constant, **inBox**, later.

The method `"xIncrement"` is an "instance method." It has an implicit formal parameter, `this`, and one explicit parameter, `dx`. When `"xIncrement"` is called with `invokevirtual`, two items are expected on the operand stack of the caller. The deeper of the two is expected to be an instance of some class and is used to select which method named `"xIncrement"` is actually run. That instance object is bound to the parameter `this` in the newly built frame and the other item on the stack is bound to the variable `dx`.

The byte code above increments the `"x"` field of `this` by the amount `dx`. Ignore for a moment the `load` instruction at 0. Instructions 1 and 2 push the contents of the `"x"` field onto the operand stack. Instruction 3 pushes `dx` and instruction 4 adds the two together, leaving the sum on the stack. The `load` instruction we ignored above, at 0, has pushed a reference to the `this` object onto the stack, now just under the sum. The `putfield` at 5 deposits the sum into the `"x"` field of that object, changing the heap. The `return` at 6 returns (no results) to the caller (i.e., this method is of return type "`void`"). It is convenient whenever we define a method to define the corresponding clock function for it. In the case of `"xIncrement"`, which consists seven primitive, non-branching byte codes, the clock function is constant and returns 8. (Our convention is that the clock for a method includes the cycle for the byte code that invokes it.)

Before discussing the `inBox` method, we consider an extension to the `"Point"` class, called the `"Colored-Point"` class. Here is the list constant denoting the TJVM declaration of that class.

ColoredPoint-class:

```
("ColoredPoint" ("Point" "Object")
                ("color")
                (setColor
                 setColorBox))
```

The class extends `"Point"` (and thus `"Object"`) and provides the new field `"color"` and two methods. The first is called `"setColor"` and is defined below. It sets the `"color"` of a `"ColoredPoint"`. The clock for this method returns 5.

setColor:

```
("setColor" (c)
            (load this)
            (load c)
            (putfield "ColoredPoint" "color")
            (return))
```

Consider the TJVM state

s_1:

```
(make-state
 (push
  (make-frame 0
              '((p . nil))
              nil
              '((new "ColoredPoint")
                (store p)
                (load p)
                (push -23)
                (invokevirtual "ColoredPoint" "xIncrement" 1)
                (load p)
                (push "Green")
                (invokevirtual "ColoredPoint" "setColor" 1)
                (load p)
                (halt)))
  nil)
 nil
 '(Point-class
   ColoredPoint-class))
```

This state is poised to execute the ten instruction program above, with one local, `p`, which is initially `nil`. The class table of the state contains both `"Point"` and its extension `"ColoredPoint"`. Inspection of the code above shows that it creates a new `"ColoredPoint"` and stores it into `p`. It then invokes `"xIncrement"` to increment the `"x"` field of `p` by -23 and invokes `"setColor"` to set the `"color"` field of `p` to `"Green"`. Of interest is the fact that the first method is in the class `"Point"` and the second is in the class `"ColoredPoint"`. The `"ColoredPoint"` `p` inherits the fields and methods of its superclass, `"Point"`.

Had the `"ColoredPoint"` class overridden the method `"xIncrement"` by including the definition of such a method, then the program above would have invoked that method rather than the one in `"Point"`, since the method is selected by searching through the superclass chain of the `this` object of the invocation, which is here `p`, an object of class `"ColoredPoint"`.

Consider, s'_1, the result of running the TJVM on s_1 for 21 steps,

s'_1:

```
(tjvm s1 21).
```

"21" is obtained by adding up the clocks for each instruction above.

Of interest is `(deref (top (stack (top-frame s'1))) (heap s'1))`. This expression dereferences the topmost item on the operand stack of s'_1, with respect to the heap of that state. The topmost item on the stack is, of course, the reference that is the value of the local `p`. That reference was created by `new` at the beginning of the program. Dereferencing it through the final heap produces the "logical meaning" of `p` at the conclusion of the program. The result is

```
(("ColoredPoint" ("color" . "Green"))
 ("Point" ("x" . -23) ("y" . 0))
 ("Object"))
```

This is an *instance* in the ACL2 semantics of the TJVM. It represents an object of class `"ColoredPoint"`. It enumerates the fields of that class and of all its superclasses and specifies the value of each field. We

see that at the conclusion of the program above the `"color"` of the object is set to `"Green"`, the `"x"` field (in the `"Point"` superclass) is set to -23 as a result of our `"xIncrement"` and the `"y"` field is set to (the "uninitialized" value) 0. The `"Object"` class in the TJVM has no fields.

5 Proving Theorems about Objects

Our `"Point"` class contains a second method, called `"inBox"`, which determines whether its `this` object is within a given rectangle in the plane. The rectangle is specified by two points, `p1` and `p2`, which are the lower-left and upper-right corners of the box. Here is the definition of the method.

inBox:

```
("inBox" (p1 p2)
         (load p1)                    ;  0
         (getfield "Point" "x")       ;  1
         (load this)                  ;  2
         (getfield "Point" "x")       ;  3
         (sub)                        ;  4
         (ifgt 21)                    ;  5
         (load this)                  ;  6
         (getfield "Point" "x")       ;  7
         (load p2)                    ;  8
         (getfield "Point" "x")       ;  9
         (sub)                        ; 10
         (ifgt 15)                    ; 11
         (load p1)                    ; 12
         (getfield "Point" "y")       ; 13
         (load this)                  ; 14
         (getfield "Point" "y")       ; 15
         (sub)                        ; 16
         (ifgt 9)                     ; 17
         (load this)                  ; 18
         (getfield "Point" "y")       ; 19
         (load p2)                    ; 20
         (getfield "Point" "y")       ; 21
         (sub)                        ; 22
         (ifgt 3)                     ; 23
         (push 1)                     ; 24
         (xreturn)                    ; 25
         (push 0)                     ; 26
         (xreturn))                   ; 27
```

This is a straightforward compilation (for the TJVM) of the following Java (which is written in a slightly awkward style to make the correspondence with the byte code more clear).

```
public boolean inBox(Point p1, Point p2){
if (p1.x <= this.x &
    this.x <= p2.x &
    p1.y <= this.y &
    this.y <= p2.y)
  {return true;}
else {return false;}}
```

We will specify and prove the correctness of this method in a moment. But we must develop some Lisp functions for dealing with points and in so doing illustrate our preferred style for dealing with TJVM objects in general, at the logical level.

Consider the two Lisp functions below for retrieving the x- and y-coordinates of a point.

```
(defun Point.x (ref s)
  (field-value "Point" "x" (deref ref (heap s))))

(defun Point.y (ref s)
  (field-value "Point" "y" (deref ref (heap s))))
```

Observe that they take the TJVM state, s, as arguments. That is because we apply them to *references* to Points, not to *instances*. Had we chosen the latter course, we would have defined `Point.x` as

```
(defun Point.x (p)
  (field-value "Point" "x" p))
```

but would have to call it by writing `(Point.x (deref ref (heap s)))`. We belabor this point for a reason: when defining the Lisp concepts necessary to speak formally about TJVM objects should we focus on references or on instances? If we focus on references, we must also have available the state, or at least the heap, with respect to which those references are to be dereferenced. This is the choice we have made and yet it seems complicated.

The problem with focusing on instances is that instances may contain references. Thus, even if we focus on instances the heap is still relevant. This does not arise with `"Point"` instances because their fields do not contain references. But consider the class `"LinkedList"`, with two fields, `"head"` and `"next"`. Suppose the `"next"` field generally contains a (reference to a) `"LinkedList"` instance. Then a typical instance of `"LinkedList"` might be:

```
(("LinkedList" ("head" . 1) ("next" . (REF 45)))
 ("Object")).
```

Since the object itself might be "circular" we cannot, in general, replace `(REF 45)` by the instance to which it dereferences. Thus, we see that while instances may be more convenient than references for simple classes like `"Point"`, they are no more convenient than references in general.

We therefore chose references because it tends to induce a uniform style in the definition of the logical functions for manipulating objects, that style being to dereference the reference with respect to the state, use the resulting instance and then recursively deal with the interior references. This means that the Lisp formalization of the semantics of a TJVM method may very closely resemble the algorithm implemented by the byte code. This is convenient because the verification of a method generally takes place in two steps. *In the first step we prove that the interpretation of the byte code produces the same TJVM state transformation as described at a higher level by the Lisp function. In the second step we prove that the Lisp function enjoys some desirable properties.* The first step is complicated by the semantics of the byte code interpreter and so it is convenient for the Lisp to be "close" to the byte code. Once we have gotten away from the byte code to the simpler applicative Lisp world, we deal with the complexity of proving that the Lisp "does the right thing."

In this paper we do not further consider instances that reference other instances.

We next consider the `"inBox"` method and focus on its "clock function." How many TJVM cycles are required to execute an invocation of that method? We can express the clock function for `"inBox"` with:

```
(defun inBox-clock (this p1 p2 s)
  (cond ((> (Point.x p1 s)
            (Point.x this s))
         9)
        ((> (Point.x this s)
            (Point.x p2 s))
         15)
        ((> (Point.y p1 s)
            (Point.y this s))
         21)
        (t 27)))
```

Despite the apparent simplicity of the `"inBox"` method, the time it takes depends not only on the inputs *this*, p_1, and p_2, but on the state in which it is called. This is unavoidable since the very meaning of references to objects change as the heap changes.

What theorem might we wish to prove about `"inBox"`? The obvious theorem is that it returns 1 or 0 (the TJVM and JVM "booleans") according to whether the `this` object is within the box defined by `p1` and `p2`. We define the Lisp function `inBox` as follows:

```
(defun inBox (this p1 p2 s)
  (and (<= (Point.x p1 s)
           (Point.x this s))
       (<= (Point.x this s)
           (Point.x p2 s))
       (<= (Point.y p1 s)
           (Point.y this s))
       (<= (Point.y this s)
           (Point.y p2 s))))
```

Observe that the Lisp `inBox` predicate is dependent upon the TJVM state with respect to which the TJVM objects are dereferenced. Again, this is unavoidable. Even though the references to the three points in question are constants, the x-y locations of the denoted points may change over time and whether a point is in a given box is, quite literally, a function of the TJVM state. (Indeed, the locations may be changed even by methods that are not directly passed these three references because other objects in the heap may contain these references.)

Before we state the correctness of the `"inBox"` method formally we do so informally. Our first statement, below, is just an approximation, modeled on the theorem we proved about `"fact"`.

> **First Approximation** Suppose s_0 is a TJVM state whose next instruction is `(invokevirtual "Point" "inBox" 2)`, where the meaning of the name `"Point"` in the class table is our **Point-class**. Let *this* be the third item on the operand stack, let p_1 be the second item, and let p_2 be the topmost item. Then the TJVM state obtained by stepping s_0 `(inBox-clock` *this* p_1 p_2 s_0`)` times is state just s_0 with the program counter incremented by one and the three items removed from the operand stack and replaced by 1 or 0 according to whether `(inBox` *this* p_1 p_2 s_0`)`. The heap is unchanged.

This approximation is inadequate in two respects. First, we are not concerned with the just the instruction `(invokevirtual "Point" "inBox" 2)` but in any instruction of the form `(invokevirtual` *class* `"inBox" 2)`. A careful reading of the semantic function for `invokevirtual` reveals that the semantics is independent of the first operand of the instruction! The actual method invoked is the one named `"inBox"` in the superclass chain of the "this" object, not the method named `"inBox"` in the class named in the `invokevirtual` instruction.[1] Therefore, rather than require that `(next-inst` s_0`)` be `(invokevirtual "Point" "inBox" 2)` we require that it be a list whose first, third and fourth elements are `invokevirtual`, `"inBox"` and 2, respectively.

The second change to this approximation is that we must insist that the method found by looking up the superclass chain of *this* be **inBox**. In particular, just because we invoke `"inBox"` in a TJVM state in which `"Point"` is defined as above, do we know that the resolved method will be **inBox**? No! The superclass chain of *this* might include a class that overrides `"inBox"` and defines it a different way. If we add the hypothesis that the resolved method is **inBox** then we may delete the hypothesis, in our approximation above, requiring that `"Point"` be as defined here: it does not matter how `"Point"` is defined as long as the resolved method is our **inBox**.

> **Theorem.** `"inBox"` is correct: Suppose s_0 is a TJVM state whose next instruction is of the form `(invokevirtual` *class* `"inBox" 2)`. Let *this* be the third item on the operand stack, let

[1] This is also true of the JVM `invokevirtual` instruction. The reason `invokevirtual` has that operand is so that in a JVM implementation the given class name can be used to find a "dispatch vector" associated with the named class to shortcut the search up the superclass chain.

p_1 be the second item, and let p_2 be the topmost item. Suppose the nearest method named `"inBox"` in the superclass chain of *this* is **inBox**. Then the TJVM state obtained by stepping s_0 `(inBox-clock` *this* p_1 p_2 s_0`)` times is state s_0 with the program counter incremented by one and the three items removed from the operand stack and replaced by 1 or 0 according to whether `(inBox` *this* p_1 p_2 s_0`)`. The heap is unchanged.

Here is the formal rendering of the theorem:

Theorem. `"inBox"` is correct:

```
(implies (and (consp (next-inst s0))
              (equal (car (next-inst s0)) 'invokevirtual)
              (equal (caddr (next-inst s0)) "inBox")
              (equal (cadddr (next-inst s0)) 2)

              (equal this (top (pop (pop (stack (top-frame s0))))))
              (equal p1  (top (pop (stack (top-frame s0)))))
              (equal p2  (top (stack (top-frame s0))))

              (equal (lookup-method "inBox"
                                    (class-name-of-ref this (heap s0))
                                    (class-table s0))
                     inBox))
         (equal
          (tjvm s0 (inBox-clock this p1 p2 s0))
          (make-state
           (push (make-frame (+ 1 (pc (top-frame s0)))
                             (locals (top-frame s0))
                             (push (if (inBox this p1 p2 s0) 1 0)
                                   (popn 3 (stack (top-frame s0))))
                             (program (top-frame s0)))
                 (pop (call-stack s0)))
           (heap s0)
           (class-table s0))))
```

This theorem is proved by straightforward symbolic evaluation, i.e., the repeated unfolding of the definition of `tjvm`, together with Boolean and arithmetic simplification.

Once proved, how is this theorem used? Suppose the theorem prover's rewriter encounters a call of `tjvm` in which the second argument is an `inBox-clock` expression. (Previously proved rules about `tjvm` will insure that arithmetic combinations of such expressions are decomposed into appropriate nests of `tjvm` expressions applied to these clock functions.) Then the rewriter tries to establish that the `next-inst` of the TJVM state is an `invokevirtual` of `"inBox"` and the resolved method named `"inBox"` in the superclass chain of the *this* object is **inBox**. These actions are caused by backchaining. The three hypotheses equating *this*, p_1, and p_2 to stack expressions are not restrictive: they are established by binding *this*, p_1, and p_2 to the corresponding expressions. Provided the hypotheses are established, the rewriter then replaces the target call of `tjvm` by a new state expression in which the program counter is advanced by one, the three actuals of `"inBox"` are removed from the stack and the appropriate value is pushed. Thus, as with our earlier theorem about `"fact"`, this theorem configures ACL2 to simplify a call of `"inBox"` almost as though it were a built-in TJVM instruction whose semantics is given in terms of the Lisp function `inBox`.

We have glossed over a minor difficulty with the application of the lemma. Note that the state s_0 occurs twice in the left-hand side of the concluding equality. In actual applications, the states matching these two occurrences of s_0 may not be identical. The first is the state of the TJVM at the time of the `invokevirtual` and the second is the state at the time the clock expression for the superior computation was computed. We handle this possibility by providing a corollary to the above rule in which the two occurrences of s_0 are given different names, s_0 and s_1, and we add an additional hypothesis requiring the proof that the `inBox-clock` of these two states be identical.

We finally turn to a theorem about a method that modifies an object, i.e., that modifies the heap. This is the second method of the `"ColoredPoint"` class:

setColorBox:

```
("setColorBox" (p1 p2 color)
               (load this)
               (load p1)
               (load p2)
               (invokevirtual "ColoredPoint" "inBox" 2)
               (ifeq 4)
               (load this)
               (load color)
               (putfield "ColoredPoint" "color")
               (return))
```

This void instance method takes three arguments, two points and a color. If the "this" object is in the box specified by the two points, the method sets the color of the "this" object to the specified color. The clock function for `"setColorBox"` is

```
(defun setColorBox-clock (this p1 p2 c s)
  (declare (ignore c))
  (++ 4
      (inBox-clock this p1 p2 s)
      (if (inBox this p1 p2 s)
          5
        2)))
```

The primary effect of invoking `"setColorBox"` is to produce a new heap. That heap is described by

```
(defun setColorBox-heap (this p1 p2 c s)
  (if (inBox this p1 p2 s)
      (let ((instance (deref this (heap s)))
            (address (cadr this)))
        (bind
         address
         (set-instance-field "ColoredPoint" "color" c instance)
         (heap s)))
    (heap s)))
```

The theorem stating the correctness of `"setColorBox"` is

Theorem. `"setColorBox"` is correct:

```
(implies (and (consp (next-inst s0))
              (equal (car (next-inst s0)) 'invokevirtual)
              (equal (caddr (next-inst s0)) "setColorBox")
              (equal (cadddr (next-inst s0)) 3)

              (equal this (top (pop (pop (pop (stack (top-frame s0)))))))
              (equal p1  (top (pop (pop (stack (top-frame s0))))))
              (equal p2  (top (pop (stack (top-frame s0)))))
              (equal color (top (stack (top-frame s0))))

              (equal (lookup-method "inBox"
                                    (class-name-of-ref this (heap s0))
                                    (class-table s0))
                     inBox)
              (equal (lookup-method "setColorBox"
                                    (class-name-of-ref this (heap s0))
                                    (class-table s0))
```

```
            setColorBox))
       (equal
        (tjvm s0 (setColorBox-clock this p1 p2 color s0))
        (make-state
         (push (make-frame (+ 1 (pc (top-frame s0)))
                           (locals (top-frame s0))
                           (popn 4 (stack (top-frame s0)))
                           (program (top-frame s0)))
               (pop (call-stack s0)))
         (setColorBox-heap this p1 p2 color s0)
         (class-table s0))))
```

This theorem is exactly analogous to the one about `"inBox"`. The effect of the theorem is to configure ACL2 so that when it sees a `tjvm` expression with the `setColorBox-clock` it increments the program counter by one, pops four things off the operand stack, and sets the heap to that described by `setColorBox-heap`.

Such a move is interesting by virtue of the following easy-to-prove lemma about that heap:

Theorem.

```
(implies (and (refp ref)
              (refp this))
         (equal (deref ref
                       (setColorBox-heap this p1 p2 color s))
                (if (and (equal ref this)
                         (inBox this p1 p2 s))
                    (set-instance-field "ColoredPoint" "color" color
                                        (deref this (heap s)))
                  (deref ref (heap s)))))
```

This theorem specifies the key property of the new heap. It defines how to dereference an arbitrary reference, *ref*, with respect to the new heap. If *ref* is the *this* object of the `"setColorBox"` invocation, and that object is within the specified box, then the dereferenced object is the result of setting the `"color"` of the same object in the old heap. Otherwise, dereferencing with respect to the new heap produces the same result as dereferencing with respect to the old heap. Thus, `setColorBox-heap` is just a succinct symbolic representation of the heap produced by `"setColorBox"` and whenever references arise with respect to it, they are eliminated and replaced by modified instances obtained through the old heap.

While we have not illustrated classes that chain instances together or instance methods that are recursive or iterative, these aspects of the TJVM should raise no problems not predicted by our handling of the recursive factorial and the simple instance methods shown. That is not to say that such proofs are trivial, only that their nature is entirely predictable from what has been shown here. The expectation is that the ACL2 user able to do the proofs shown would be able to develop the techniques necessary to do proofs of this more involved nature.

6 Conclusion

We have shown how a simple version of the Java Virtual Machine can be formalized in the applicative Common Lisp supported by the ACL2 theorem prover. We have discussed how ACL2 can be configured to make it straightforward to reason about static and instance methods, including recursive methods and methods which modify objects in the heap.

The use of "clock functions" to characterize the length of TJVM computations follows a standard paradigm in interpreter-based proofs. Clock expressions allow the user to structure the proof. Every addend in a `++`-nest gives rise to a single call of `tjvm` which must be simplified. Thus, by choice of an appropriate clock expression one can decompose proofs into the segments about which one is prepared to reason by specially-tailored lemmas. To the extent that one can decompose a computation mechanically into such regions, one can mechanically generate clock expressions to control the proof decomposition. Clock expressions

compose in the obvious way with `invokevirtual` and `invokestatic`. Iterative computations are handled similarly.

While both clock expressions and the Lisp functions describing the semantics of TJVM methods might appear complicated, it is crucial to remember the compositional nature of specifications and proofs. Once the clock expression for a method has been written and the corresponding specification has been proved, it is not necessary to think about or reveal in subsequent proofs the details of the methods used as "subroutines."

The use of clocks encourages a focus on terminating computations. However, by adding a flag to the TJVM state indicating whether the computation halted normally or "ran out of time" it is possible to phrase partial correctness results in this framework, in which theorems have the additional hypothesis of "provided n is sufficient to guarantee termination." Furthermore, one can use explicit clocks, as we do, but address oneself to non-terminating computations and still characterize the state produced at certain points in the infinite computation.

Our toy JVM ignores many aspects of the JVM, as noted earlier, including the omission of many byte code instructions, the finiteness of resources, error handling, exceptions, and multiple threads. Many of these aspects could be incorporated into a formal model. Some, such as the inclusion of additional byte codes, would not affect the complexity of proofs at all. The others would preserve the basic character of the theorems and proofs described here but, in some cases, would require the explicit statement of additional hypotheses to insure the sufficiency of available resources and the absence of errors. New proof machinery would have to be created (via the proofs of suitable lemmas) to enable ACL2 to reason about TJVM computations in which exceptions are thrown or multiple threads are used.

Our experience, notably that reported in Young's dissertation, [13] and in the author's work on Piton [11], is that when dealing with resource limitations and exceptions it is best to produce several layers of abstraction, each formally related to the next by lemmas, one of which is free of those concepts and corresponds to our `tjvm`. That is, we see a model like our TJVM as being a component of a stack of abstract machines that enables formal discussion of programming language concepts not included in `tjvm`.

The most problematic aspect of our formalization may appear to be the fact that TJVM objects cannot be represented by (the non-circular) objects of applicative Common Lisp, necessitating the "reference versus instance" debate summarized here and the explicit provision of the heap in our specification functions. It cannot be denied that this is a complication. Similar problems have been dealt with in Flatau's dissertation [7]. However, we do not think this can be avoided simply because it reflects the underlying reality of object oriented programming.

7 Acknowledgments

I am especially grateful to Rich Cohen, who patiently explained his ACL2 model of his "defensive" Java Virtual Machine, upon which my TJVM is modeled. I am also very grateful to the undergraduates at UT to whom I have taught the TJVM, as well as my teaching assistant for that course last year, Pete Manolios.

References

[1] W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. Special Issue on System Verification. *Journal of Automated Reasoning*, 5(4):409–530, December, 1989.

[2] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press: New York, 1979.

[3] R. S. Boyer and J S. Moore. Mechanized Formal Reasoning about Programs and Computing Machines. In R. Veroff (ed.), *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos,* MIT Press, 1996.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*, Academic Press: London, 1997.

[5] B. Brock, M. Kaufmann and J S. Moore, "ACL2 Theorems about Commercial Microprocessors," in M. Srivas and A. Camilleri (eds.) *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275–293, 1996.

[6] R. M. Cohen, *The Defensive Java Virtual Machine Specification, Version 0.53*, Electronic Data Systems, Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701 (email: cohen@aus.edsr.eds.com).

[7] A. D. Flatau, *A verified implementation of an applicative language with dynamic storage allocation*, PhD Thesis, University of Texas at Austin, 1992.

[8] M. Kaufmann and J Strother Moore "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp," *IEEE Transactions on Software Engineering*, **23**(4), pp. 203–213, April, 1997

[9] M. Kaufmann and J Strother Moore "A Precise Description of the ACL2 Logic," http://-www.cs.utexas.edu/users/moore/publications/km97a.ps.Z, April, 1998.

[10] T. Lindholm and F. Yellin *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[11] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Automated Reasoning Series, Kluwer Academic Publishers, 1996.

[12] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[13] W. D. Young, *A Verified Code-Generator for a Subset of Gypsy*, PhD Thesis, University of Texas at Austin" 1988.

[14] Y. Yu. *Automated Proofs of Object Code For a Widely Used Microprocessor.* PhD thesis, University of Texas at Austin, 1992. Lecture Notes in Computer Science, Springer-Verlag (to appear). ftp://ftp.cs.utexas.edu/pub/techreports/tr93-09.ps.Z.