Guide to The dJVM Model
Version 0.5 Alpha

** DRAFT **

Richard M. Cohen

May 12, 1997

Abstract

This is a brief guide to running the Defensive Java Virtual Machine (dJVM) model in ACL2. It describes the input format for classes, and the subset of the JVM instructions supported in version 0.5 Alpha. A tutorial example illustrates loading class definitions into the model and running them.

Contents

Introduction

This guide provides a brief introduction to the defensive Java Virtual Machine version 0.5 Alpha (hereafter dJVM 0.5). Readers interested in the details of the model should refer to *The Defensive Java Virtual Machine Specification (version 0.5)* available from the Computational Logic web page (at `http://www.cli.com`). Readers interested in quickly running examples on the dJVM 0.5 model should look at the tutorial example in section 5, page 13.

1 What is the Defensive Java Virtual Machine?

The *Defensive Java Virtual Machine* (dJVM) is Java Virtual Machine that includes enough run-time checks to assure type-safe execution of programs without recourse to a bytecode verifier, as is required by the standard Java Virtual Machine. If a program attempts to perform an unsafe operation, the dJVM detects the error and signals the error appropriately. This usually means that the machine halts with an error flag.

The dJVM 0.5 model is a preliminary definition of the dJVM built in a mathematical logic called ACL2. As well as being a formal logic, ACL2 also has a mechanical theorem prover. The logic is based on the applicative (or functional) subset of Common Lisp, and offers the ability to execute functions defined in the logic using an underlying Common Lisp implementation.

The Alpha release of the dJVM 0.5 model is a very rough, first draft of a dJVM model. It has been assembled quickly, and has not been tuned for execution speed, for ease of use, or for easy use of mechanical proof checking. But is does run some JVM programs, and some theorem have been proven about it.

The dJVM 0.5 supports 102 of the 202 standard JVM bytecode instructions. It includes:

- Invoking instance methods, class methods, instance initialization methods and returning from them
- Invoking overridden instance methods from a superclass
- Instance creation and initialization
- Accessing values in and storing values into instance fields
- `int` and `long` integer data and operations
- Branching instructions

The dJVM 0.5 does leave out several extremely useful features of the JVM. These features (with the exception of floating point) are all candidates to be added to the dJVM in the future.

- Arrays

- Interfaces
- Throwing and handling exceptions
- Dynamic loading of classes
- Concurrency, multithreading, and synchronization
- Floating Point data and operations

2 Overview of Running the Model

The model is defined in the ACL2 dialect of Common Lisp [Steele Jr., 1984]. An pre-built executable image for Solaris 2 systems is available. (See section B, page 38.)

The top-level of the model is a defensive interpreter for the Java Virtual Machine bytecodes. This interpreter will run JVM bytecoded methods defined in symbolic representations of Java class class files. Currently there is not a mechanical tool available to construct this representation directly from standard Java class files. But this representation is easily constructed from the output of the `javap` class-file printer, distributed with the Java Developers Kit (JDK).

Section 3 describes the format of the symbolic class declarations. Section 4 lists the JVM instructions supported and the symbolic formats of the instructions. Section 5 gives an example of converting class files to the dJVM format, loading them into the dJVM, running the interpreter, and inspecting the dJVM state.

2.1 Class-file converter

The class-file converter prepares an image of a class file for execution by the dJVM. This is a Common Lisp program. As such it lacks any formal specifications, and is not described here. Its use is described in the *Guide to The dJVM Model*.

To run it:

```
> djvm0.5
...
ACL2 Version 1.8 built January 13, 1996  15:24:42.
Initialized with dJVM 0.38 of 4/18/1997.
...

ACL2 !>:q

Exiting the ACL2 read-eval-print loop.  To re-enter, execute (LP).
ACL2>(convert-class-file "Point")
"Point.lisp"
```

This produces a new file, `Point.lisp`, containing the converted class definition, which can be loaded into the dJVM. Figure 1 shows the resulting file.

3 Class Declarations

Here's a simple Java class declaration.

```
class Point extends Object {
  int x, y;

  public int x() {
    return this.x;
    }

  public void move (int x, int y) {
    this.x = x;
    this.y = y;
    return;
    }
}
```

After compiling this with the standard Java compiler from JavaSoft's JDK 1.1, we use `javap` to print the class file. The output from `javap` is shown in figure 2. As we shall see, the dJVM representation of class `Point` will look quite similar to this. In particular the format of dJVM method declarations is a simple transformation from the format displayed by `javap`.

As we shall see below, the form that `javap` uses to display methods is very similar to that used by the dJVM.

3.1 Format of Method Declarations

The dJVM declaration forms for the two methods in class `Point` are show below in figures 3 and 4. The method declaration is constructed using the dJVM function `make-java-method`. That function takes keyword arguments which give the various attributes of a JVM method declaration. The method declarations show below simply reflect the information present in the `javap` output. Again the type signatures of the methods have been converted to the class-file format.

Each keyword argument takes a single value. The order of keyword arguments does not affect the value of the expression.

Details of Method Declarations

Here are detailed explanations of the keyword arguments to the method declaration constructor `make-java-method`. In ACL2 (as in Common Lisp) keywords begin with a colon, for example, `:name`.

```
;; dJVM class file constructed from Point.class
;;
;; [Converted on 4/22/1997 at 4:53 p.m.
;;  using Make-dJVM-Class version 0.5 alpha 1]

(in-package "ACL2")
(set-verify-guards-eagerness 2)

(defun class-Point ()
 (MAKE-CLASS-DECL
      :NAME "Point"
      :ACCESS-FLAGS 'NIL
      :SUPERCLASS "java.lang.Object"
      :SUPERCLASSES NIL
      :SURROGATE (REF-TO-NULL )
      :INTERFACES NIL
      :FIELDS (LIST (MAKE-FIELD :NAME "x"
                                :SIG "I"
                                :PROTECTION ':DEFAULT-PROTECTION
                                ... )
                    (MAKE-FIELD :NAME "y"
                                ... ))
      :METHODS (LIST (MAKE-JAVA-METHOD :NAME "x"
                                       :SIG "()I"
                                       :CLASS-NAME "Point"
                                       :PROTECTION ':PUBLIC
                                       :ACCESS-FLAGS 'NIL
                                       :MAX-STACK 1
                                       :MAX-LOCALS 1
                                       :BODY '((0 ALOAD_0)
                                               (1 GETFIELD "Point" "x" "I")
                                               (4 IRETURN)
                                               )
                                       :EXCEPTION-TABLE NIL
                                       :ATTRS NIL)
                                       ... )
      :ATTRS '(("SourceFile" "Point.java")
               ))

) ;; end of class Point

(thm (class-decl-p (class-Point)))

;;
;; End of File
;;
```

Figure 1: Converted version of class Point

```
class Point extends java.lang.Object
    /* ACC_SUPER bit set */

    int x;
    int y;
    public int x();
        /* Stack=1, Locals=1, Args_size=1 */
    public void move(int,int);
        /* Stack=2, Locals=3, Args_size=3 */
    Point();
        /* Stack=1, Locals=1, Args_size=1 */

Method int x()
   0 aload_0
   1 getfield #4 <Field Point.x I>
   4 ireturn

Method void move(int,int)
   0 aload_0
   1 iload_1
   2 putfield #4 <Field Point.x I>
   5 aload_0
   6 iload_2
   7 putfield #5 <Field Point.y I>
  10 return

Method Point()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object.<init>()V>
   4 return
```

Figure 2: Output from javap

```
(make-java-method :name "x"
                  :class-name "Point"
                  :sig "()I"
                  :access-flags '()
                  :protection ':public
                  :max-stack 1
                  :max-locals 1
                  :body '((0 aload 0)
                          (1 getfield "Point" "x" "I")
                          (4 ireturn))
                  :exception-table nil
                  :attrs nil)
```

Figure 3: Declaration for Method `x`

```
(make-java-method :name "move"
                  :class-name "Point"
                  :sig  "(II)V"
                  :access-flags '()
                  :protection ':public
                  :max-stack 2
                  :max-locals 1
                  :body '((0 aload 0)
                          (1 iload 1)
                          (2 putfield "Point" "x" "I")
                          (5 aload 0)
                          (6 iload 2)
                          (7 putfield "Point" "y" "I")
                          (10 return))
                  :exception-table nil
                  :attrs nil)
```

Figure 4: Declaration for Method `move`

```
(make-java-method :name "<init>"
                  :class-name "Swill"
                  :sig "()V"
                  :access-flags '()
                  :protection ':public
                  :max-stack 1
                  :max-locals 1
                  :body '((0 aload_0)
                          (1 invokespecial "java.lang.Object" "<init>" "()V")
                          (4 return))
                  :exception-table nil
                  :attrs nil))
```

Figure 5: Declaration for `Point` Method `<init>`

:name the simple name of the method, as a string.

:class-name the name of the class in which this method is being declared. (It is convenient to be able to identify the class of a method. So we record this information as part of the method declaration.)

:sig the type signature of the method. This uses the format prescribed for class files. The format is "*(T*)R*", where T stands for an argument type signature (as described below for field declarations), and R is the result type-signature of the method. The return-type may be a field type-signature or the letter `V`, which denotes a `void` method that does not return a value.

:access-flags one of `:static`, `:final`, `:synchronized`, `:native`, or `:abstract`.

:protection one of `:public`, `:private`, `:protected`, or `:default-protection`.

:max-stack the maximum stack size (in words).

:max-locals the maximum number of local variables used (in words).

:body the bytecode body of the method. The bytecode instructions are represented symbolically.

:exception-table the exception table for the method. The dJVM 0.5 Alphadoes not support exceptions. So this should always be `nil`.

:attrs other attributes associated with the method. The dJVM 0.5 Alphadoes not honor any attributes. So this should always be `nil`.

3.2 Format of Field Declarations

The field declarations for the two fields in class `Point` are shows in figure 6. The default protection attribute has been made explicit, and the field type has been translated into the JVM internal format.

The internal format for type signatures of fields is:

- `I` for `int`.
- `J` for `long`.
- `Lxyz;` for a reference to class `xyz`.

The dJVM 0.5 Alpha does not support any field attributes.

```
(make-field :access-flags '()
            :name "x"
            :sig "I"
            :protection ':default-protection
            :attrs nil)

(make-field :access-flags '()
            :name "y"
            :sig "I"
            :protection ':default-protection
            :attrs nil)
```

Figure 6: Field Declarations for `Point`

3.3 Format of Class Declarations

The complete declaration for class `Point` is given in figure 7. The actual method declarations are abbreviated by defining functions to construct the dJVM method declarations show above. The function constructing the method declaration for `Point.move` is shown in figure 8.

4 Instructions & Instruction Formats

The dJVM supports 103 of the JVM instructions.

Here is a list of the supported instructions and their dJVM symbolic format. In this list the following place-holder names are used:

- `index` — 8-bit unsigned integer
- `wide-index` — 16-bit unsigned integer
- `byte-constant` — 8-bit signed integer

```
(defun class-decl-for-point ()
  (make-class-decl :name          "Point"
                   :surrogate     (make-tv :ref 0)
                   :status        'initialized
                   :access-flags  '(:public)
                   :superclass    "java.lang.Object"
                   :superclasses  '("java.lang.Object")
                   :interfaces    nil
                   :fields        (list (make-field :access-flags '()
                                                    :name "x"
                                                    :sig "I"
                                                    :protection ':default-protection
                                                    :attrs nil)
                                        (make-field :access-flags '()
                                                    :name "y"
                                                    :sig "I"
                                                    :protection ':default-protection
                                                    :attrs nil))
                   :methods       (list (point-x-method)
                                        (point-move-method))
                   :attrs         nil
                   ))
```

Figure 7: Class Declaration for `Point`

```
(defun point-move-method ()
  (make-java-method :name "move"
                    :class-name "Point"
                    :sig  "(II)V"
                    :access-flags '()
                    :protection ':public
                    :max-stack 2
                    :max-locals 1
                    :body '((0 aload 0)
                            (1 iload 1)
                            (2 putfield "Point" "x" "I")
                            (5 aload 0)
                            (6 iload 2)
                            (7 putfield "Point" "y" "I")
                            (10 return))
                    :exception-table nil
                    :attrs nil))
```

Figure 8: Function Constructor for Method `Point.move`

- `short-constant` — 16-bit signed integer
- `offset` — 16-bit signed offset
- `wide-offset` — 32-bit signed offset
- `"Class"` — a class name
- `"field"` — a field name
- `"method"` — a method name
- `"I"` — a field signature (e.g., `int`)
- `"(II)J"` — a method signature (e.g., taking arguments `(int,int)` and returning `long`)

1. `(aconst_null)`
2. `(aload index)`
3. `(aload_0)`
4. `(aload_1)`
5. `(aload_2)`
6. `(aload_3)`
7. `(aload_wide wide_index)`
8. `(areturn)`
9. `(astore index)`
10. `(astore_0)`
11. `(astore_1)`
12. `(astore_2)`
13. `(astore_3)`
14. `(astore_wide wide-index)`
15. `(bipush byte-constant)`
16. `(dup)`
17. `(dup2)`
18. `(dup2_x1)`
19. `(dup_x1)`
20. `(dup_x2)`
21. `(getfield "Class" "field" "I")`

22. (getstatic "Class" "field" "I")
23. (goto offset)
24. (goto_w wide-offset)
25. (i2l)
26. (i2s)
27. (iadd)
28. (iand)
29. (iconst_0)
30. (iconst_1)
31. (iconst_2)
32. (iconst_3)
33. (iconst_4)
34. (iconst_5)
35. (iconst_m1)
36. (idiv)
37. (if_acmpeq offset)
38. (if_acmpne offset)
39. (if_icmpeq offset)
40. (if_icmpge offset)
41. (if_icmpgt offset)
42. (if_icmple offset)
43. (if_icmplt offset)
44. (if_icmpne offset)
45. (ifeq offset)
46. (ifge offset)
47. (ifgt offset)
48. (ifle offset)
49. (iflt offset)
50. (ifne offset)
51. (ifnonnull offset)
52. (ifnull offset)
53. (iinc index byte-constant)
54. (iload index)
55. (iload_0)
56. (iload_1)
57. (iload_2)
58. (iload_3)
59. (iload_wide wide-index)
60. (imul)
61. (ineg)
62. (invokespecial "Class" "method" "(II)J")
63. (invokestatic "Class" "method" "(II)J")
64. (invokevirtual "Class" "method" "(II)J")
65. (ior)
66. (ireturn)
67. (istore index)
68. (istore_0)
69. (istore_1)
70. (istore_2)
71. (istore_3)
72. (isub)
73. (l2i)
74. (ladd)
75. (land)
76. (lcmpg)
77. (lconst_0)
78. (lconst_1)
79. (ldiv)
80. (lload index)
81. (lload_0)
82. (lload_1)
83. (lload_2)
84. (lload_3)
85. (lload_wide wide-index)
86. (lookupswitch ...)
87. (lstore index)
88. (lstore_0)
89. (lstore_1)
90. (lstore_2)
91. (lstore_3)
92. (lstore_wide wide-index)
93. (new "Class")
94. (nop)
95. (pop)
96. (pop2)
97. (putfield "Class" "field" "I")
98. (putstatic "Class" "field" "I")
99. (return)
100. (sipush short-constant)
101. (swap)
102. (tableswitch ...)

5 Running the dJVM & Inspecting the State

This section shows an example of:

- converting class files into the dJVM format
- loading the converted files into the dJVM model

- running the `main` method and examining the dJVM state along the way.

There are two Java classes used in this example: class `Lbreak2`, shown in figure 9, and class `Int_Array_10`, shown in figure 10

This example was run using the dJVM 0.5 Alpha 1 model running in ACL2 version 1.8.

This example was run using an executable image of the dJVM model using ACL2 version 1.8 built on GNU Common Lisp (GCL). Most of the example is performed in ACL2, but a few places (particularly the discussion of input prompts) illustrates behavior that is GCL-specific. If you build the dJVM model using ACL2 built on a different underlying Common Lisp implementation, the form of the prompts may differ.

```
// Test instanceof
// Labelled break and continue Statements

public class Lbreak2 {

    private static void cleanup (Int_Array_10 x, int a, int b) {
      x.set(a, b);
    }

    private static boolean check (int p) {
        if (p == 8) return true;
        else return false;
    }

    public static void main () {
        int p, i = 8;
        int total = 0;
        Int_Array_10 a = new Int_Array_10 (6, 1, 2, 4, 3, 5, 8, 7, 9, 0);
        Lbreak2 lb = new Lbreak2();

test:   if (Lbreak2.check(i)) {
//            try {
               for (int j =0; j < 10; j++) {
                   if (j > i) {
                     Lbreak2.cleanup (a, j, (int)11);
                     break test;
                   }
                   if (a.elt(j) == 4) {
                     i = 6;
                     a.set(2, 111);
                     continue;
                   }
                   a.set(j, 1111);
                }
//            }
//            finally { Lbreak2.cleanup (a, i, (int)7); }
        }

        Lbreak2.cleanup (a, (int)9, (int)7);

        for (p = 0; p < a.length(); p++) {
//            System.out.println("a[" + p + "] = " + a[p]);
            total += a.elt(p);
//            System.out.println("total = " + total);
        }
        if (total != 5697) Fail.genError();
        if (lb instanceof Lbreak2) Fail.genError();
    }
}
```

Figure 9: Class Lbreak2

```
public class Int_Array_10 {

  static int a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;

  // This should trigger generation of an explicit class initialization.

  static int a9 = 1;

  // Constructors.  One with initial values.

  public Int_Array_10 () {};

  public Int_Array_10 (int x0, int x1, int x2, int x3, int x4,
                       int x5, int x6, int x7, int x8, int x9) {

    a0 = x0;      a5 = x5;
    a1 = x1;      a6 = x6;
    a2 = x2;      a7 = x7;
    a3 = x3;      a8 = x8;
    a4 = x4;      a9 = x9;
    return;
  }

  // Return the length of the array.  In our case, it is always 10.

  public int length () {
    return 10;
  }

  // Accessor method

  public int elt (int index) {
    switch (index) {
      case 0: return a0;        case 5: return a5;
      case 1: return a1;        case 6: return a6;
      case 2: return a2;        case 7: return a7;
      case 3: return a3;        case 8: return a8;
      case 4: return a4;        case 9: return a9;
      default: return 0;
      }
  }

  // Alterant method

  public void set (int index, int value) {
    switch (index) {
      case 0: a0 = value; break;        case 5: a5 = value; break;
      case 1: a1 = value; break;        case 6: a6 = value; break;
      case 2: a2 = value; break;        case 7: a7 = value; break;
      case 3: a3 = value; break;        case 8: a8 = value; break;
      case 4: a4 = value; break;        case 9: a9 = value; break;
      default: return;
      }
    return;
  }
}
```

Figure 10: Class `Int_Array_10`

5.1 Recognizing various input prompts

The standard ACL2 command prompt is `ACL2 !>`.

If you cause an "error" in ACL2, ACL2 will enter the low-level debugger. You will see a message such as

```
Correctable error: Console interrupt.
Signalled by LP.
If continued: Type :r to resume execution, or :q to quit to top level.
Broken at COND.  Type :H for Help.
ACL2>>
```

The prompt `ACL2>>` indicates that control has passed to the GCL debugger. You can exit from the GCL debugger and return the the ACL2 command level via the debugger command `:q` (for quit). When you give this command, you should get the standard ACL2 prompt again.

```
ACL2>>:q
ACL2 !>
```

At the ACL2 command level, the `:q` command exits from the ACL2 command loop and passed control to the GCL command loop. You can then type a GCL or Lisp command, such as the expression `(+ 1 2 3)` as shown below.

```
ACL2 !>:q

Exiting the ACL2 read-eval-print loop.  To re-enter, execute (LP).
ACL2>(+ 1 2 3)
6
```

You can subsequentally give the command `(lp)` to reenter the ACL2 command loop. Normally ACL2 command and function names are not case-sensitive. So you can type `(LP)`, `(lp)`, or even `(Lp)`. When you reenter the ACL2 command loop, ACL2 will print its greeting banner, and then print its standard input prompt.

```
ACL2>(LP)

ACL2 Version 1.8.  Level 1.  Cbd "/cli/project/os/java/Model/djvm0.39/".
Type :help for help.

ACL2 !>
```

5.2 Example of Running the dJVM 0.5 Model

```
% ./djvm0.50.sunos-5.i86pc
GCL (GNU Common Lisp)  Version(2.2) Tue Nov 28 19:05:01 CST 1995
Licensed under GNU Public Library License
Contains Enhancements by W. Schelter
```

```
Loading init.lsp
Finished loading init.lsp

 ACL2 Version 1.8 built November 28, 1995  23:41:03.
 Initialized with dJVM 0.50 of 5/5/1997.
 See :doc note8 for recent changes.

 NOTE!!  Proof trees are disabled in ACL2.  To enable them in emacs,
 look under the ACL2 source directory in interface/emacs/README.doc;
 and, to turn on proof trees, execute :START-PROOF-TREE in the ACL2
 command loop.   Look in the ACL2 documentation under PROOF-TREE.

ACL2 Version 1.8.  Level 1.  Cbd "/cli/project/os/java/Model/djvm0.39/".
Type :help for help.

ACL2 !>(in-package "ACL2")
 "ACL2"
```

5.3 Converting CLass Files

We must drop out of the ACL2 command loop to run the class-file converter, because the converter is written in raw Common Lisp. We do this via the `:q` (for "quit") command to ACL2. This exits from the ACL2 command loop and begins the command loop of the underlying Common Lisp system.

```
ACL2 !>:q

Exiting the ACL2 read-eval-print loop.  To re-enter, execute (LP).
```

We first convert the class `Int_Array_10`. This will read the class file `"Int_Array_10.class"` and write the dJVM image of that class to `"Int_Array_10.lisp"`.

```
ACL2>(convert-class-file "tests/Int_Array_10")
"tests/Int_Array_10.lisp"
```

Now convert the class `Lbreak2`.

```
ACL2>(convert-class-file "tests/Lbreak2")
Warning: The instruction INSTANCEOF is not implemented in the dJVM 0.5 model.
"tests/Lbreak2.lisp"
```

The class-file converter issues a warning message if it notices an instruction not supported by the dJVM 0.5 model. The `wide` instruction is not handled, and may cause spurious warning messages from the converter as well as producing a class definition that will not run in the dJVM model.

We now return the the ACL2 command loop by calling the function `lp`. ACL2 prints the greeting banner and prompts for a command.

```
ACL2>(lp)

ACL2 Version 1.8.  Level 1.  Cbd "/cli/project/os/java/Model/djvm0.39/".
Type :help for help.
```

5.4 Loading Class Files

Now we load the converted class files. First we load the class Int_Array_10 from the converted file Int_Array_10.lisp. The standard ACL2 function for loading source files is "`ld`". The `ld` function reads each form or command from the file and evaluates it. In this first example of using `ld` each command in the file will be printed before it is evaluated, followed by any output from ACL2's evaluation of the command.

The `ld` function operates essentially by calling the ACL2 command loop recursively with the file as the input stream. ACL2 indicates this "second level" of the command loop by printing the "prompt" with a double ">" (i.e. as "ACL2 !>>").

```
ACL2 !>(ld "tests/Int_Array_10.lisp" :ld-pre-eval-print t)

ACL2 Version 1.8.  Level 2.  Cbd "/cli/project/os/java/Model/djvm0.39/".
Type :help for help.

ACL2 !>>(IN-PACKAGE "ACL2")
 "ACL2"
ACL2 !>>(SET-VERIFY-GUARDS-EAGERNESS 2)
 2
ACL2 !>>(DEFUN
         CLASS-INT_ARRAY_10 NIL
         (MAKE-CLASS-DECL
          :NAME "Int_Array_10"
          :ACCESS-FLAGS '(:SUPER :PUBLIC)
          :SUPERCLASS "java.lang.Object"
          :SUPERCLASSES NIL
          :SURROGATE (REF-TO-NULL)
          :INTERFACES NIL
          :FIELDS (LIST (MAKE-FIELD :NAME "a0" :SIG
                            "I" :PROTECTION ':DEFAULT-PROTECTION
                            :ACCESS-FLAGS '(:STATIC)
                            :ATTRS NIL)
                        ⋮
                        )
          :METHODS (LIST (MAKE-JAVA-METHOD :NAME "<init>"
                                           :SIG "()V"
                                           :CLASS-NAME "Int_Array_10"
                                           :PROTECTION ':PUBLIC
                                           :ACCESS-FLAGS 'NIL
```

```
                                            :MAX-STACK 1
                                            :MAX-LOCALS 1
                                            ...)
                              ⋮
                              )
        :ATTRS '(("SourceFile" "Int_Array_10.java"))))
```

The following output is ACL2's commentary as it processes this command.

```
Since CLASS-INT_ARRAY_10 is non-recursive, its admission is trivial.
We could deduce no constraints on the type of CLASS-INT_ARRAY_10.

The guard conjecture for CLASS-INT_ARRAY_10 is trivial to prove.  CLASS-
INT_ARRAY_10 is compliant with Common Lisp.

Summary
Form:  ( DEFUN CLASS-INT_ARRAY_10 ...)
Rules: NIL
Warnings:  None
Time:  0.06 seconds (prove: 0.00, print: 0.00, other: 0.06)
 CLASS-INT_ARRAY_10

   ⋮

ACL2 !>>Bye.
 :EOF
```

This printed the full class definition as it appears in the file, along with the transcript of any proofs required to admit the class definition into the ACL2 logic. If we don't want to see this output (as we usually don't), we can use the `ld-quietly` function. (It is loaded into ACL2 as part of the miniscule user-interface support for the dJVM.)

Using `ld-quietly` if the file loads successfully, ACL2 will print `:EOF` and then prompt for another command. If there is an error loading the file, ACL2 will print `:ERROR` instead of `:EOF`. Depending on the sort of error encountered, ACL2 may print an error message before printing `:ERROR`. For more complete information on the nature of the error and where it occurred in the file, use the raw call to `ld` as shown above. (You may have to exit from the debugger (via `:q`) before issuing the `ld` command. You can tell whether control has passed into the debugger by looking at the input prompt, as described in section 5.1, page 17.)

```
ACL2 !>(ld-quietly "tests/Lbreak2.lisp")

ACL2 Version 1.8.  Level 2.  Cbd "/cli/project/os/java/Model/djvm0.39/".
Type :help for help.

 :EOF
```

5.5 Defining a dJVM State

We now define a Lisp function to construct an initial dJVM state with the classes `Int_Array_10` and `Lbreak2` loaded. Loading the converted class files defined the functions `Class-Int_Array_10` and `Class-Lbreak2`. These function names are derived mechanically from the class names, so we can deduce the function names for any converted class file.

We simply compose calls to `Djvm-Load-Class-Decl` for each class we want to load into the state. The value 99 is given in the calls indicates the maximum number of instructions to let the class initialization method run. `Int_Array_10` has a class initialization method, but it runs to completion in fewer than 99 instructions. You can give a larger number here to be sure your call initialization has sufficent time to finish.

```
ACL2 !>(defun lbreak2-djvm ()
  (Djvm-Load-Class-Decl (Class-lbreak2)
                        99
                        (Djvm-Load-Class-Decl (Class-Int_Array_10)
                                              99
                                              (set-djvm-status ':running
                                                               (Initial-Djvm)))))
```

ACL2 now processes the function definition. Before accepting the definition, ACL2 will attempt to prove that the function always terminates and that the guards for all functions called in its body are satisfied. If ACL2 cannot successfully complete these proofs, it will not accept the function definition. Normally the transcript of its proof attempt is printed to the terminal. Here's a brief example.

```
Since LBREAK2-DJVM is non-recursive, its admission is trivial.  We
observe that the type of LBREAK2-DJVM is described by the theorem
(CONSP (LBREAK2-DJVM)).  We used the :type-prescription rule DJVM-LOAD-
CLASS-DECL.

The non-trivial part of the guard conjecture for LBREAK2-DJVM is

Goal
(AND
  (WEAK-DJVM-P (INITIAL-DJVM))
  (CLASS-DECL-P (CLASS-INT_ARRAY_10))
  (DJVM-P (SET-DJVM-STATUS :RUNNING (INITIAL-DJVM)))
  (CLASS-DECL-P (CLASS-LBREAK2))
  (DJVM-P (DJVM-LOAD-CLASS-DECL (CLASS-INT_ARRAY_10)
                                99
                                (SET-DJVM-STATUS :RUNNING (INITIAL-DJVM))))).

But we reduce the conjecture to T, by the :executable-counterparts
of INITIAL-DJVM, WEAK-DJVM-P, CLASS-INT_ARRAY_10, CLASS-DECL-P, SET-
DJVM-STATUS, DJVM-P, CLASS-LBREAK2, DJVM-LOAD-CLASS-DECL and IF.
```

```
Q.E.D.

That completes the proof of the guard theorem for LBREAK2-DJVM.  LBREAK2-
DJVM is compliant with Common Lisp.

Summary
Form:  ( DEFUN LBREAK2-DJVM ...)
Rules: ((:EXECUTABLE-COUNTERPART INITIAL-DJVM)
        (:EXECUTABLE-COUNTERPART WEAK-DJVM-P)
        (:EXECUTABLE-COUNTERPART CLASS-INT_ARRAY_10)
        (:EXECUTABLE-COUNTERPART CLASS-DECL-P)
        (:EXECUTABLE-COUNTERPART SET-DJVM-STATUS)
        (:EXECUTABLE-COUNTERPART DJVM-P)
        (:EXECUTABLE-COUNTERPART CLASS-LBREAK2)
        (:EXECUTABLE-COUNTERPART DJVM-LOAD-CLASS-DECL)
        (:EXECUTABLE-COUNTERPART IF)
        (:TYPE-PRESCRIPTION DJVM-LOAD-CLASS-DECL))
Warnings:  None
Time:  0.17 seconds (prove: 0.08, print: 0.01, other: 0.08)
 LBREAK2-DJVM
```

Let's make sure that our function really constructs a dJVM state.

```
ACL2 !>(djvm-p (lbreak2-djvm))
T
```

Let's check that the state has initialized and is running normally. If the class initialization failed to terminate or completed abnormally, the status will not be "`:running`".

We can interogate the status by explicitly looking at the djvm-status field.

```
ACL2 !>(djvm-status (lbreak2-djvm))
:RUNNING
```

We can also express this as a putative theorem for the ACL2 theorem prover to verify. Since the theorem prover can run our compiled Lisp functions when given concrete (i.e., non-symbolic) expressions, this is about as fast as the query above.

```
ACL2 !>(thm (equal (djvm-status (lbreak2-djvm)) ':running))

But we reduce the conjecture to T, by the :executable-counterparts
of LBREAK2-DJVM, DJVM-STATUS and EQUAL.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: ((:EXECUTABLE-COUNTERPART LBREAK2-DJVM)
        (:EXECUTABLE-COUNTERPART DJVM-STATUS)
        (:EXECUTABLE-COUNTERPART EQUAL))
```

```
Warnings:  None
Time:  0.18 seconds (prove: 0.16, print: 0.00, other: 0.02)

Proof succeeded.
```

5.6 Running the dJVM

Show the initial static variables of `Int_Array_10`. At this point they should all be zero.

```
ACL2 !>(show-object 3 (lbreak2-djvm))
(3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 1)
          ("a8" :INT 0)
          ("a7" :INT 0)
          ("a6" :INT 0)
          ("a5" :INT 0)
          ("a4" :INT 0)
          ("a3" :INT 0)
          ("a2" :INT 0)
          ("a1" :INT 0)
          ("a0" :INT 0))
   (:STATUS LOADED)
   (:LOCK NIL)
   (:LOADER (:REF 0)))
```

We now run the dJVM from the initial state, stopping just before returning from `Int_Array_10.<init>(IIIIIIIIII)V`, so that we can see that the instance initializer's parameters have been recoreedd in the class' static variables.

(It may sound odd to record data from an instance initialization in static variables, but that is how `Int_Array_10` is defined.)

We can get there by specifying to run for 40 steps, or until we hit that PC.

The basic way to run the dJVM model is to use the function Run-Class-Main (defined in initial-djvm.lisp). This function takes a class name, a clock value (i.e., a step count), and a dJVM state. It runs the public, static main method of the specified class for a maximum of "clock" steps.

However, the function `Run-Main`, defined in `show-fns.lisp`, is useful when runningg the dJVM interactively. The function `Run-Main` has the additional property that when the machine stops with a non-empty call-stack, then the local varible 999 of the current-call frame will hold the number of instructions executed. Thus it is clear if the machine completed or halted before exhausting the clock value. Using `Run-Main` you can specify that the public, static main method should be executed until a given `pc` value is encountered.

Here's an example using the step count.

```
ACL2 !>(show-stack (run-main "Lbreak2" (+ 40) (lbreak2-djvm)))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Int_Array_10")
                 (:METHOD (:FULL-NAME "Int_Array_10.<init>(IIIIIIIIII)V")
```

```
                         (:PROTECTION :PUBLIC)
                         (:ACCESS-FLAGS)
                         (:BODY ----------
                               (48 PUTSTATIC "Int_Array_10" "a9" "I")
                               (51 RETURN)
                               ----------))
                 (:CIA 48)
                 (:PC 51)
                 (:LOCALS (0 :REF 5)
                          (1 :INT 6)
                          (2 :INT 1)
                          (3 :INT 2)
                          (4 :INT 4)
                          (5 :INT 3)
                          (6 :INT 5)
                          (7 :INT 8)
                          (8 :INT 7)
                          (9 :INT 9)
                          (10 :INT 0)
                          (999 :INT 40))
                 (:STACK)
                 (:OBJECT-REF (:REF 5))
                 (:NEW-REFS))
          (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                (23 INVOKESPECIAL
                                    "Int_Array_10" "<init>" "(IIIIIIIII)V")
                                (26 ASTORE_3)
                                (27 NEW "Lbreak2")
                                ----------))
                 (:CIA 23)
                 (:PC 26)
                 (:LOCALS (1 :INT 8) (2 :INT 0))
                 (:STACK (:REF 5))
                 (:OBJECT-REF (:REF 0))
                 (:NEW-REFS (:REF 5))))
```

Here's an example specifying the `pc` to stop at. We must still specify a step-count. So we just give a large value and count on the `:to-pc` argument to stop execution. Of course if the given `pc` is not encountered before the step-count is exhausted, `Run-Main` will still terminate after than many steps.

```
ACL2 !>(show-stack (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 51))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Int_Array_10")
                 (:METHOD (:FULL-NAME "Int_Array_10.<init>(IIIIIIIII)V")
                          (:PROTECTION :PUBLIC)
```

```
                              (:ACCESS-FLAGS)
                              (:BODY ----------
                                     (48 PUTSTATIC "Int_Array_10" "a9" "I")
                                     (51 RETURN)
                                     ----------))
                     (:CIA 48)
                     (:PC 51)
                     (:LOCALS (0 :REF 5)
                              (1 :INT 6)
                              (2 :INT 1)
                              (3 :INT 2)
                              (4 :INT 4)
                              (5 :INT 3)
                              (6 :INT 5)
                              (7 :INT 8)
                              (8 :INT 7)
                              (9 :INT 9)
                              (10 :INT 0)
                              (999 :INT 40))
                     (:STACK)
                     (:OBJECT-REF (:REF 5))
                     (:NEW-REFS))
              (FRAME (:CLASS "Lbreak2")
                     (:METHOD (:FULL-NAME "Lbreak2.main()V")
                              (:PROTECTION :PUBLIC)
                              (:ACCESS-FLAGS :STATIC)
                              (:BODY ----------
                                     (23 INVOKESPECIAL
                                         "Int_Array_10" "<init>" "(IIIIIIIII)V")
                                     (26 ASTORE_3)
                                     (27 NEW "Lbreak2")
                                     ----------))
                     (:CIA 23)
                     (:PC 26)
                     (:LOCALS (1 :INT 8) (2 :INT 0))
                     (:STACK (:REF 5))
                     (:OBJECT-REF (:REF 0))
                     (:NEW-REFS (:REF 5))))
```

We can look at the array values, and see that they have changes as the dJVM runs.

```
ACL2 !>(show-object 3 (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 51))
(3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 0)
          ("a8" :INT 9)
          ("a7" :INT 7)
          ("a6" :INT 8)
          ("a5" :INT 5)
          ("a4" :INT 3)
          ("a3" :INT 4)
```

```
            ("a2" :INT 2)
            ("a1" :INT 1)
            ("a0" :INT 6))
     (:STATUS LOADED)
     (:LOCK NIL)
     (:LOADER (:REF 0)))
```

Since we will be running Lbreak2.main many times, it is convenient to define a little function so that we don't have to type the full expression each time. We'll define a function that just takes a step count and applies it to running Lbreak2.main.

```
ACL2 !>(defun run-lbreak (n)
  (declare (xargs :guard (integerp n)))
  (if (>= n 0)
      (run-main "Lbreak2" n (lbreak2-djvm))
    nil))

Since RUN-LBREAK is non-recursive, its admission is trivial.  We could
deduce no constraints on the type of RUN-LBREAK.

The non-trivial part of the guard conjecture for RUN-LBREAK is

Goal
(IMPLIES (AND (INTEGERP N) (<= 0 N))
         (DJVM-P (LBREAK2-DJVM))).

But we reduce the conjecture to T, by the :executable-counterparts
of LBREAK2-DJVM and DJVM-P.

Q.E.D.

That completes the proof of the guard theorem for RUN-LBREAK.  RUN-
LBREAK is compliant with Common Lisp.

Summary
Form:  ( DEFUN RUN-LBREAK ...)
Rules: ((:EXECUTABLE-COUNTERPART LBREAK2-DJVM)
        (:EXECUTABLE-COUNTERPART DJVM-P)
        (:DEFINITION NOT))
Warnings:  None
Time:  0.26 seconds (prove: 0.05, print: 0.00, other: 0.21)
 RUN-LBREAK
```

Let's use it to look at the array values again.

```
ACL2 !>(show-object 3 (run-lbreak 40))
(3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 0)
          ("a8" :INT 9)
          ("a7" :INT 7)
```

```
          ("a6" :INT 8)
          ("a5" :INT 5)
          ("a4" :INT 3)
          ("a3" :INT 4)
          ("a2" :INT 2)
          ("a1" :INT 1)
          ("a0" :INT 6))
   (:STATUS LOADED)
   (:LOCK NIL)
   (:LOADER (:REF 0)))
```

One more instruction will get us passed the return instruction in the method `Int_Array_10.<init>` and just about to execute the instruction after the `invokespecial` call to it. Note that the `cia` register in the frame points to the invokespecial instruction, because that was the last instruction executed in this frame. The `pc` register points to the following instruction, since that is the next one to be executed in this frame (now that the method `Int_Array_10.<init>(IIIIIIIII)V` has completed).

```
ACL2 !>(show-stack (run-lbreak (+ 41)))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                 (23 INVOKESPECIAL
                                     "Int_Array_10" "<init>" "(IIIIIIIII)V")
                                 (26 ASTORE_3)
                                 (27 NEW "Lbreak2")
                                 ----------))
                 (:CIA 23)
                 (:PC 26)
                 (:LOCALS (1 :INT 8)
                          (2 :INT 0)
                          (999 :INT 41))
                 (:STACK (:REF 5))
                 (:OBJECT-REF (:REF 0))
                 (:NEW-REFS)))
```

Now we'll stop execution just after allocating an instance of `Lbreak2`. Note that a reference to the instance appears in the `new-refs` of the current frame, indicating that the instance is considered to be uninitialized in the context of this frame.

```
ACL2 !>(show-stack (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 30))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
```

```
                              (:ACCESS-FLAGS :STATIC)
                              (:BODY ---------- (27 NEW "Lbreak2")
                                     (30 DUP)
                                     (31 INVOKESPECIAL "Lbreak2" "<init>" "()V")
                                     ----------))
                     (:CIA 27)
                     (:PC 30)
                     (:LOCALS (1 :INT 8)
                              (2 :INT 0)
                              (3 :REF 5)
                              (999 :INT 43))
                     (:STACK (:REF 6))
                     (:OBJECT-REF (:REF 0))
                     (:NEW-REFS (:REF 6))))
```

Here's the new instance.

```
ACL2 !>(show-object 6 (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 30))
(6 INSTANCE (:CLASS (:REF 4))
   (:DATA ("Lbreak2") ("java.lang.Object"))
   (:LOCK NIL))
```

If we run a little further the instance (at heap address 6) is considered to be initialized (within this frame), as evidenced by its absence from `new-refs`.

```
ACL2>(show-stack (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 34))
(:STATUS= :RUNNING
 :STACK= (FRAME (:CLASS "Lbreak2")
          (:METHOD (:FULL-NAME "Lbreak2.main()V")
                   (:PROTECTION :PUBLIC)
                   (:ACCESS-FLAGS :STATIC)
                   (:BODY ----------
                          (31 INVOKESPECIAL "Lbreak2" "<init>" "()V")
                          (34 ASTORE 4)
                          (36 ILOAD_1)
                          ----------))
          (:CIA 31)
          (:PC 34)
          (:LOCALS (1 :INT 8) (2 :INT 0) (3 :REF 5) (999 :INT 49))
          (:STACK (:REF 6))
          (:OBJECT-REF (:REF 0))
          (:NEW-REFS)))
```

The compiler generated the loop-test at the bottom of the loop, and branches to it from the top the first time through. The test is at instruction 105. Stopping at `pc` = 103 has the loop variable (j) as the top element of the stack. The first time through the loop we note that j=0. (The value of j is stored as local variable number 5).

```
ACL2 !>(show-stack (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 103))
(:STATUS= :RUNNING
```

```
 :STACK=  (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                 (101 ILOAD 5)
                                 (103 BIPUSH 10)
                                 (105 IF_ICMPLT -56)
                                 ----------))
                 (:CIA 101)
                 (:PC 103)
                 (:LOCALS (1 :INT 8)
                          (2 :INT 0)
                          (3 :REF 5)
                          (4 :REF 6)
                          (5 :INT 0)
                          (999 :INT 62))
                 (:STACK (:INT 0))
                 (:OBJECT-REF (:REF 0))
                 (:NEW-REFS)))
```

After executing the loop once, the value of j should be 1, and the first element of the array (static variable `a0` in class `Int_Array_10`) should be set to 1111. Let's look at the stack.

```
ACL2 !>(show-stack (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 103 :ntimes 2))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                 (101 ILOAD 5)
                                 (103 BIPUSH 10)
                                 (105 IF_ICMPLT -56)
                                 ----------))
                 (:CIA 101)
                 (:PC 103)
                 (:LOCALS (1 :INT 8)
                          (2 :INT 0)
                          (3 :REF 5)
                          (4 :REF 6)
                          (5 :INT 1)
                          (999 :INT 87))
                 (:STACK (:INT 1))
                 (:OBJECT-REF (:REF 0))
                 (:NEW-REFS)))
```

And let's check that the first element of the array (static variable `a0` in class `Int_Array_10`) has the value 1111.

```
ACL2 !>(show-object 3 (run-main "Lbreak2" 100 (lbreak2-djvm) :to-pc 103 :ntimes 2))
(3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 0)
          ("a8" :INT 9)
          ("a7" :INT 7)
          ("a6" :INT 8)
          ("a5" :INT 5)
          ("a4" :INT 3)
          ("a3" :INT 4)
          ("a2" :INT 2)
          ("a1" :INT 1)
          ("a0" :INT 1111))
   (:STATUS LOADED)
   (:LOCK NIL)
   (:LOADER (:REF 0)))
```

Since we're going to want to look at the stack and object 3 many times, we will define a macro that shows both values. This time we define a macro, rather than a function, because in ACL2 macro calls can take optional keyword parameters. Thus we can call the macro `show-lbreak` defined below with just a clock argument, or give both a clock argument and a `:to-pc` keyword argument, or with a clock argument and both a `:to-pc` keyword argument and a `:ntimes` keyword argument.

```
ACL2 !>(defmacro show-lbreak (clock &key to-pc ntimes)
  `(let ((new-djvm (run-main "Lbreak2" ,clock (lbreak2-djvm)
                             :to-pc ,to-pc :ntimes ,ntimes)))
     (list (show-stack new-djvm)
           (show-object 3 new-djvm))))

Summary
Form:  ( DEFMACRO SHOW-LBREAK ...)
Rules: NIL
Warnings:  None
Time:  0.02 seconds (prove: 0.00, print: 0.00, other: 0.02)
 SHOW-LBREAK
```

If you are running the dJVM model as interpreted code, rather than as compiled code, you may have to expand the GCL stack to accomocate this computation. This is because the GCL compiler converts tail-recursive functions into iterative functions. This eliminates the function-call overhead and the run-time stack space required for a recursive call. If you are running the model as interpreted Lisp code, this optimization has not been done, and each dJVM instruction step consumes some of the GCL stack. In the standard GCL image, 1000 steps is more than can be handled interpretively.

In order to increase the GCL stack, you must exit from the ACL2 command loop via

```
:q
```

and then execute the following command at the GCL command level:

```
(setq SYSTEM:*MULTIPLY-STACKS* 4)
```

This will cause the a garbage collection, and then ACL2/DJVM start-up banner will be printed again after the stack size has been increased.

The loop-exit branch corresponding to `break test` in the Java program (page 15) is at address 63. Let's stop there, and see what the array looks like.

```
ACL2 !>(show-lbreak 1000 :to-pc 63)
((:STATUS= :RUNNING
  :STACK=  (FRAME (:CLASS "Int_Array_10")
                  (:METHOD (:FULL-NAME "Int_Array_10.elt(I)I")
                           (:PROTECTION :PUBLIC)
                           (:ACCESS-FLAGS)
                           (:BODY ----------
                                  (60 GETSTATIC "Int_Array_10" "a1" "I")
                                  (63 IRETURN)
                                  (64 GETSTATIC "Int_Array_10" "a2" "I")
                                  ----------))
                  (:CIA 60)
                  (:PC 63)
                  (:LOCALS (0 :REF 5)
                           (1 :INT 1)
                           (999 :INT 98))
                  (:STACK (:INT 1))
                  (:OBJECT-REF (:REF 0))
                  (:NEW-REFS))
           (FRAME (:CLASS "Lbreak2")
                  (:METHOD (:FULL-NAME "Lbreak2.main()V")
                           (:PROTECTION :PUBLIC)
                           (:ACCESS-FLAGS :STATIC)
                           (:BODY ----------
                                  (69 INVOKEVIRTUAL
                                      "Int_Array_10" "elt" "(I)I")
                                  (72 ICONST_4)
                                  (73 IF_ICMPNE 16)
                                  ----------))
                  (:CIA 69)
                  (:PC 72)
                  (:LOCALS (1 :INT 8)
                           (2 :INT 0)
                           (3 :REF 5)
                           (4 :REF 6)
                           (5 :INT 1))
                  (:STACK)
                  (:OBJECT-REF (:REF 0))
                  (:NEW-REFS)))
 (3 A-CLASS (:NAME "Int_Array_10")
```

```
   (:DATA ("a9" :INT 0)
          ("a8" :INT 9)
          ("a7" :INT 7)
          ("a6" :INT 8)
          ("a5" :INT 5)
          ("a4" :INT 3)
          ("a3" :INT 4)
          ("a2" :INT 2)
          ("a1" :INT 1)
          ("a0" :INT 1111))
   (:STATUS LOADED)
   (:LOCK NIL)
   (:LOADER (:REF 0))))
```

Oops! We hit instruction 63 in `Int_Array_10.elt`! But that instruction is only executed when fetching array element 1. So it will only be encountered once in the "test" loop. So we want to give the keyword argument `:ntimes 2` in the call to `show-lbreak` to get passed this point.

```
ACL2 !>(show-lbreak 1000 :to-pc 63 :ntimes 2)
((:STATUS= :RUNNING
  :STACK=  (FRAME (:CLASS "Lbreak2")
                  (:METHOD (:FULL-NAME "Lbreak2.main()V")
                           (:PROTECTION :PUBLIC)
                           (:ACCESS-FLAGS :STATIC)
                           (:BODY ----------
                                  (60 INVOKESTATIC "Lbreak2"
                                      "cleanup" "(LInt_Array_10;II)V")
                                  (63 GOTO 45)
                                  (66 ALOAD_3)
                                  ----------))
                  (:CIA 60)
                  (:PC 63)
                  (:LOCALS (1 :INT 6)
                           (2 :INT 0)
                           (3 :REF 5)
                           (4 :REF 6)
                           (5 :INT 7)
                           (999 :INT 259))
                  (:STACK)
                  (:OBJECT-REF (:REF 0))
                  (:NEW-REFS)))
 (3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 0)
          ("a8" :INT 9)
          ("a7" :INT 11)
          ("a6" :INT 1111)
          ("a5" :INT 1111)
          ("a4" :INT 1111)
          ("a3" :INT 4)
```

```
            ("a2" :INT 111)
            ("a1" :INT 1111)
            ("a0" :INT 1111))
     (:STATUS LOADED)
     (:LOCK NIL)
     (:LOADER (:REF 0))))

ACL2 !>(show-lbreak 1000 :to-pc 103 :ntimes 8)
((:STATUS= :RUNNING
  :STACK=  (FRAME (:CLASS "Lbreak2")
                  (:METHOD (:FULL-NAME "Lbreak2.main()V")
                           (:PROTECTION :PUBLIC)
                           (:ACCESS-FLAGS :STATIC)
                           (:BODY ----------
                                  (101 ILOAD 5)
                                  (103 BIPUSH 10)
                                  (105 IF_ICMPLT -56)
                                  ----------))
                  (:CIA 101)
                  (:PC 103)
                  (:LOCALS (1 :INT 6)
                           (2 :INT 0)
                           (3 :REF 5)
                           (4 :REF 6)
                           (5 :INT 7)
                           (999 :INT 240))
                  (:STACK (:INT 7))
                  (:OBJECT-REF (:REF 0))
                  (:NEW-REFS)))
 (3 A-CLASS (:NAME "Int_Array_10")
    (:DATA ("a9" :INT 0)
           ("a8" :INT 9)
           ("a7" :INT 7)
           ("a6" :INT 1111)
           ("a5" :INT 1111)
           ("a4" :INT 1111)
           ("a3" :INT 4)
           ("a2" :INT 111)
           ("a1" :INT 1111)
           ("a0" :INT 1111))
    (:STATUS LOADED)
    (:LOCK NIL)
    (:LOADER (:REF 0))))
```

Instruction 144 is the test whether the sum total of the array is 5697. Let's stop there and see what's on the operand stack.

```
ACL2 !>(show-stack (run-main "Lbreak2" 1000 (lbreak2-djvm) :to-pc 144))
(:STATUS= :RUNNING
 :STACK=  (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
```

```
                    (:PROTECTION :PUBLIC)
                    (:ACCESS-FLAGS :STATIC)
                    (:BODY ----------
                           (141 SIPUSH 5697)
                           (144 IF_ICMPEQ 6)
                           (147 INVOKESTATIC "Fail" "genError" "()V")
                           ----------))
           (:CIA 141)
           (:PC 144)
           (:LOCALS (1 :INT 6)
                    (2 :INT 5697)
                    (3 :REF 5)
                    (4 :REF 6)
                    (5 :INT 7)
                    (0 :INT 10)
                    (999 :INT 455))
           (:STACK (:INT 5697)
                   (:INT 5697))
           (:OBJECT-REF (:REF 0))
           (:NEW-REFS)))
```

Now we just run the dJVM until it stops. It will stop because the `instance-of` instruction has not yet been implemented in the this version of the dJVM model. The status field shows that the machine has halted, because it is not `:running`, and the value shows why the machine has halted.

```
ACL2 !>(show-stack (run-main "Lbreak2" 1000 (lbreak2-djvm)))
(:STATUS= (:ERROR "Unrecognized instruction"
                  (INSTANCEOF "Lbreak2"))
          :STACK=
          (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                 (152 INSTANCEOF "Lbreak2")
                                 (155 IFEQ 6)
                                 ----------))
                 (:CIA 152)
                 (:PC 152)
                 (:LOCALS (1 :INT 6)
                          (2 :INT 5697)
                          (3 :REF 5)
                          (4 :REF 6)
                          (5 :INT 7)
                          (0 :INT 10)
                          (999 :INT 458))
                 (:STACK (:REF 6))
                 (:OBJECT-REF (:REF 0))
                 (:NEW-REFS)))
```

ACL2 allows us to assign to variables at the top-level of the command loop using the function assign.

```
ACL2 !>(assign x '(alpha bravo charlie))
 (ALPHA BRAVO CHARLIE)
```

We can then refer to the varibles in the top-level forms using the function `@` applied to the variable name. Let's check the value of the variable x.

```
ACL2 !>(@ x)
(ALPHA BRAVO CHARLIE)
```

However, we note that when we use `assign` as a top-level function, the new value of the variable is printed. The dJVM state with `Lbreak2` and `Int_Array_10` loaded is ~400 lines when printed. So we want to avoid printing this needlessly. The `assign*` macro has the same effect as the `assign` macro, but it returns the variable's name instead of the variable's new value.

```
ACL2 !>(assign* y '(delta echo))
 Y
```

We can now assign the a dJVM state to the variable `LB` without having to see the entire state printed out.

```
ACL2 !>(assign* LB (RUN-MAIN "Lbreak2" 1000 (LBREAK2-DJVM) :TO-PC 103 :NTIMES 10))
 LB
```

Now we can look at just the parts of that state that we want to see. Recall that normally Lisp and ACL2 ignore the case of function names and variable names in input. So we can type either `LB` or `lb` below and get the same result.

```
ACL2 !>(show-stack (@ lb))
(:STATUS= (:ERROR "Unrecognized instruction"
                  (INSTANCEOF "Lbreak2"))
          :STACK=
          (FRAME (:CLASS "Lbreak2")
                 (:METHOD (:FULL-NAME "Lbreak2.main()V")
                          (:PROTECTION :PUBLIC)
                          (:ACCESS-FLAGS :STATIC)
                          (:BODY ----------
                                 (152 INSTANCEOF "Lbreak2")
                                 (155 IFEQ 6)
                                 ----------))
                 (:CIA 152)
                 (:PC 152)
                 (:LOCALS (1 :INT 6)
                          (2 :INT 5697)
                          (3 :REF 5)
                          (4 :REF 6)
                          (5 :INT 7)
                          (0 :INT 10)
```

```
                       (999 :INT 458))
                  (:STACK (:REF 6))
                  (:OBJECT-REF (:REF 0))
                  (:NEW-REFS)))
```

And we can look at several parts without having to recompute the state.

```
ACL2 !>(show-object 3 (@ lb))
(3 A-CLASS (:NAME "Int_Array_10")
   (:DATA ("a9" :INT 7)
          ("a8" :INT 9)
          ("a7" :INT 11)
          ("a6" :INT 1111)
          ("a5" :INT 1111)
          ("a4" :INT 1111)
          ("a3" :INT 4)
          ("a2" :INT 111)
          ("a1" :INT 1111)
          ("a0" :INT 1111))
   (:STATUS LOADED)
   (:LOCK NIL)
   (:LOADER (:REF 0)))
```

The dJVM 0.5 model does not support the `instanceof` instruction, but we can make that test by explicitly using a function from the dJVM interpreter. The dJVM 0.5 model includes the predicate `instance-of-class-p`. It simply lacks the instruction version of the test. The function takes 4 arguments: an object, a class name, the heap, and the class table.

Let's see whether it says the object at heap address 6 is really an instance of the class `Lbreak2`...

```
ACL2 !>(let ((djvm (@ lb)))
       (instance-of-class-p  (deref '(:ref 6) (djvm-heap djvm))
                             "Lbreak2"
                             (djvm-heap djvm)
                             (djvm-class-table djvm)))
T
```

Now let's exit from ACL2 back to GCL...

```
ACL2 !>:q

Exiting the ACL2 read-eval-print loop.  To re-enter, execute (LP).
```

and then exit from GCL back to the Unix shell. If your version of ACL2 is built upon a different implementation of Common Lisp, the function to exit from Lisp will probably be different.

```
ACL2>(lisp:bye)
%
```

This concludes the dJVM 0.5 tutorial.

A Summary of User-Level Functions

This section gives a summary of the functions typically used to run the dJVM 0.5 Alphamodel. Since the model is implemented in ACL2, and ACL2 is a language of pure functions, each of these functions takes the dJVM state as an argument and returns a value, usually a new dJVM state.

- `(djvm-step djvm)`

  This function executes one instruction in the dJVM given state, `djvm`.

  The argument `djvm` must be a dJVM state with a non-empty call-stack, and the top-most call-frame must contain a bytecoded method. (Otherwise ACL2 will report a run-time guard violation.)

  The return value of the function will be a new dJVM state with the result of attempting to execute the next instruction. As well as any error-halts caused by the individual instruction, `djvm-step` itself will report if the initial state has an invalid `PC` value or if the next instruction is not one defined in the dJVM 0.5 Alphamodel.

- `(djvm-run n-steps djvm)`

  This function calls `djvm-step` on the state `djvm`, and then on the resulting state, and so on, until `n-steps` instructions have been executed (or the error flag is set).

- `(Initial-Djvm)` – constructs an initial dJVM state containing the classes `class` and `object`.

- `(djvm-load-class-decl (class-decl n-steps djvm)`

  This function adds the class to the class table, and runs its method `<clinit>` method if it has one. `n-steps` gives the maximum number of steps to run.

  This is not a real class-loader in the common Java sense.

- `(Run-Class-Main class-name n-steps djvm)`

  This function will run the method `main` in the named class (if it is declared `public static void`). `n-steps` gives the maximum number of steps to run.

- `(run-main class-name n-steps djvm &key to-pc ntimes)`

  This is not part of the dJVM model itself. This is part of a small, crude package for testing and debugging the model and observing program execution. [See the file `show-fns.lisp`.]

- `(show-frame djvm)`

  Displays the current-frame. The body of the current method is abbreviated.

- `(show-stack djvm)`

  Displays all frames in the current call-stack.

- `(show-heap djvm)`

  Displays all objects in the heap.

- `(show-object addr djvm)`

  Displays one object from the heap.

- `(run-to-pc n-steps to-pc n-times djvm)`

  Runs the dJVM starting in state `djvm` until either `n-steps` steps have been taken or until the `PC` register has taken on the value `top-pc` exactly `n-times` times or until execution halts, whichever occurs first.

B How to Obtain dJVM and ACL2

They are both available via anonymous FTP from `ftp.cli.com`.

B.1 The dJVM 0.5 Distribution

The full distribution includes:

- executable images of the dJVM 0.5 model for Solaris 2 systems on both Sparc-based systems and x86-based systems,
- source files describing the model as executable ACL2,
- the draft technical report explaining the model,
- examples,
- associated auxilliary files (e.g., a makefile, etc.)
- this guide.

B.2 The ACL2 Distribution

The ACL2 v1.8 distribution was used to build the public version of dJVM 0.5. It is currently available from `ftp.cli.com`. ACL2 version 1.9 is expected to be publicly released shortly.

References

[Bevier, 1995] William R. Bevier.
Tools for describing typed data structures in acl2.
Technical report, Computational Logic, Inc., September 1995.

[Kaufmann and Moore, 1994] Matt Kaufmann and J Strother Moore.
Design goals of acl2.
Technical Report 101, Computational Logic, Inc., August 1994.

[Kaufmann and Moore, 1996] M. Kaufmann and J S. Moore.
ACL2: An Industrial Strength Version of Nqthm.
In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.
Revised version to appear in IEEE Trans. on Software Engineering, 1997.

[R.S. Boyer, 1991] J S. Moore R.S. Boyer, M.J. Kaufmann.
A short note on some advantages of acl2.
Technical Report 215, Computational Logic, Inc., 1991.

[Steele Jr., 1984] Guy L. Steele Jr.
Common LISP: The Language.
Digital Press, 1984.