JSVM: A implementing a JVM in javascript

Alexandre Passos and Vijayaraghavan Apparsundaram

October 17, 2011

This paper describes a prototype java virtual machine implemented in javascript. We used the rhino implementation of javascript, which runs on top of the JVM and gives access to all the java standard libraries plus JVM debuggers and profilers. The implementation fits in a single source file with around 1300 lines of code.

1 Overall approach

The main goal guiding this implementation was to avoid doing as much work as possible, at the penalty of performance and strict observation of the specification. We reasoned that it was acceptable to sacrifice performance because everything is already running under the many overheads of javascript, and by making the core simple this interpreter can be easily extended to correctly run any desired program.

The interpreter is structured as follows. A big array of javascript functions (`INST`) is maintained. This array is indexed by bytes and in each byte there is a javascript function responsible for executing that bytecode. This function has access to the local scope, the current class, the code block, the program counter, and the stack, and it is responsible to modify the stack and local variables (and global state, such as static fields) as it sees fit, returning in the end either another setting for the program counter or one of a few special tokens. The code block is kept internally as-is (essentially a list of bytes) and each instruction is responsible for accessing any extra operands it needs from the code block. While this decision enhanced practicality it makes it annoying to implement the "wide" bytecode, which is left as future work in our implementation. These bytecode functions are allowed to call the interpreter recursively.

The code is structured essentially like a C program, and objects are only used to represent JVM classes and objects, rather than organizing the interpreter code itself.

Exceptions are handled by the throwing bytecode returning a token indicating that an exception has been thrown. Then, the interpreter walks through the handlers, executing them as appropriate, and if no handler is found it returns "throw" so interpreters higher up the stack can handle the exception. try..finally blocks are handled by the java compiler, and the virtual machine only has to implement the "jsr" and "ret" instructions.

In our implementation both classes and objects have vtables assigned to them, and these vtables are responsible for virtual and interface method resolution.

The java standard library was mostly left unimplemented, and we have javascript code for class stubs for only four basic classes: Object, Throwable, Serializable, and Exception. These classes are not fully implemented. As we're running on top of the JVM, when possible we redirect access to fields and methods in unknown classes to appropriate fields and methods in the base JVM. This means programs can use System libraries, but, for example, cannot place objects inside collections or pass nonprimitive objects to standard library methods. The indirection is handled by the javascript function eval which is implemented using java reflection libraries.

Even though we were not concerned with performance ease of implementation dictated that most bytecodes operate in constant time. Some nonconstant operations are type casting (and, hence, exception handling), switch statements (which are repeatedly parsed every time they are executed), and multidimensional array initialization.

1.1 Individual contributions

Alexandre wrote the classfile parser and made the initial architectural decisions about where to base the implementation, how to organize the interpreter, which data structures to use, etc. Vijayaraghavan implemented many of the instructions, including those responsible for type checking, arrays, and switch. He also did most of the testing and support infrastructure.

2 Key challenges

The hardest challenge in implementing a JVM in javascript is performance, which we admitedly decided to ignore. This, however, freed us to use higher-order functions to implement sets of highly similar bytecodes only once. Not worrying about compactness of in-memory datastructures also freed us to implement everything as a bunch of hash tables.

Another challenge is that the specification presupposes that the in-memory data structures used to store code and data are tightly packed, and hence requires special code for the case where it isn't. Instructions such as dup_x2, for example, which have different behavior for doubles versus floats, are implemented incorrectly. This issue also complicated the classfile parsing, which on the other hand was simpler than it would be otherwise due to our reliance on the java standard libraries.

3 Adaptations to javascript (and other purposeful incompatibilities)

There are many incompatibilities between the JVM spec and our implementations. We stuck as much as possible to the javascript data model, so internally our JVM does not do as much as it could to differentiate between data types. We also did not implement classfile validation and anything else which somehow verified whether the compiler was performing its job correctly.

TODO: add more

4 Example sections

Our main interpreter loop is as follows:

```
function runMethod(method, cls, locals) {
    var stack = []
   if (method.jscode) {
        // call a js version of the method, if exists
        return method.jscode(method, cls, locals)
    } else {
        var code = method.code
        var ip = 0
        while (1) {
            var res = INST[code[ip]](code, ip, stack, cls, locals)
            if (res == "return") {
                return "void"
            } else if (res == "ireturn") {
                return stack.pop()
            } else if (res == "throw") {
                var ex = stack.pop()
                var handlers = method.handtable
                var found = false
```

```
                for (var i = 0; i < handlers.length; ++i) {
                    if (!((ip > handlers[i].start)&&(ip < handlers[i].end))) continue
                    if (! matchType(ex[1].cls, handlers[i].type)) continue
                    ip = handlers[i].handler
                    stack.push(ex)
                    print(" --- debug -- going to handler at instr "+ip)
                    found = true
                    break
                }
                if (found) continue
                // all else failing we throw it one level up
                locals[0] = ex
                return "throw"
            } else {
                ip = res
            }
        }
    }
}
```

Using higher-order functions, we implemented bytecodes sometimes generally, as follows':

```
function iconst(i) {
    return function(c,p, s, cls, l) {
        s.push(["int", i])
        return p+1
    }
}
```

5 Numbers and limitations

TODO: put some numbers here