

The SML-JVM Toolkit

Version 0.5, February 3, 1998

This paper presents the ‘SML-JVM toolkit’, a set of utilities for generating Java bytecode and class files[1, 3]. The SML-JVM toolkit is implemented as a set of Standard ML structures, data types and functions[4, 5].

Currently, the SML-JVM toolkit does *not* perform complete bytecode verification on generated classes. In other words, it is possible to generate class files that may not be accepted by Sun’s bytecode verifier.

1 Toolkit Modules

These are the modules in the SML-JVM toolkit:

Classdecl	abstract class declaration
Jvmtype	JVM data types
Label	abstract labels (used for branch targets, etc.)
Localvar	abstract local variable indices
Bytecode	abstract bytecode instructions
Stackdepth	calculation of maximum operand stack depth (written by Peter Sestoft)
Constpool	abstract constant pool representation
Emitcode	conversion from abstract bytecode instructions to binary representation
Classfile	concrete class file structure

2 Class Representation

In the SML-JVM toolkit a Java class or interface is represented as a collection of field and method declarations. A class/interface declaration also specifies a set of ‘access’ flags, a direct superclass (except for class Object, the superclass of all other classes), a set of direct superinterfaces, and a set of attributes:

```
type class_decl =
  {flags:  access_flag list,   (* access flags *)
   this:  jclass,             (* this class/interface *)
   super: jclass option,      (* direct superclass *)
   ifcs:  jclass list,        (* direct superinterfaces *)
   fdecls: field_decl list,   (* field declarations *)
   mdecls: method_decl list, (* method declarations *)
   attrs: attribute list}
```

A Java interface declaration is very similar to a class declaration; an interface simply carries the flag `ACCinterface`. Furthermore, the access flags specify whether the class/interface is public, final, and/or abstract.

An ‘attribute’ can be one of a number of standard attributes, e.g. specifying the source file of a class, but may also include non-standard information in arbitrary binary format. This permits the use of very flexible attributes for classes/interfaces, fields and methods. In general, a JVM must recognize the standard attributes defined in *The Java Virtual Machine Specification*[3], and must silently ignore any non-standard attributes that it does not recognize.

For each field declared in a class or interface the field name, type, access flags and attributes are specified:

```
type field_decl =
  {flags:  access_flag list, (* access flags *)
   name:  string,           (* unqualified name *)
   ty:    jtype,           (* field type *)
   attrs: attribute list}  (* field attributes *)
```

The field access flags specify whether the field is public, private or protected, and whether it is static, final, volatile, and/or transient. The field attributes may specify a constant value for the field, or may provide non-standard information about the field (in arbitrary format, as for classes/interfaces).

For example, a field declaration corresponding to the Java declaration

```
static final int answer = 42;
```

could be declared like this using the SML-JVM toolkit:

```
val x : field_decl =
  {flags = [ACCfinal, ACCstatic],
   name  = "answer",
   ty    = Tint,
   attrs = [CONSTVAL (Cint (Int32.fromInt 42))]}
```

Similarly, a method declaration specifies the method name, signature, access flags and attributes:

```
type method_decl =
  {flags:  access_flag list, (* access flags *)
   name:  string,           (* unqualified name *)
   msig:  method_sig,      (* method signature *)
   attrs: attribute list}  (* method attributes *)
```

The method signature specifies a list of parameter types and an optional return type; NONE means that the method does not return any value (a.k.a. void).

The method access flags specify whether the method is public, private or protected, and whether it is static, final, synchronized, native, and/or abstract. The method attributes may specify the Java bytecode for the method (if it is implemented by the class/interface), and may specify a set of exceptions that the method is declared to throw. Again, the method attributes may also include non-standard information about the method (in arbitrary format, as for classes/interfaces and fields).

For example, an ‘empty’ method `main` corresponding to the Java declaration

```
public static void main (String[] args) {}
```

could be declared like this using the SML-JVM toolkit:

```
val main : method_decl =
  {flags = [ACCpublic, ACCstatic],
   name = "main",
   msig = ([Tarray (Tclass String)], NONE),
   attrs = [CODE {stack = 0,
                  locals = 1,
                  code = [Jreturn],
                  hdl = [],
                  attrs = []}]
}
```

Note that the `CODE` attribute includes a `stack` field, specifying the maximum operand stack depth for the method, and a `locals` field specifying the maximum number of local variables used by the method. The function `maxdepth` in the `Stackdepth` module (written by Peter Sestoft) can be used to calculate the maximum stack depth for the bytecodes of a method. Similarly, the function `Localvar.count` can be used to calculate the maximum number of local variables used by a method.

The `class_decl`, `field_decl`, and `method_decl` data types are defined in the `Classdecl` module of the SML-JVM toolkit.

3 Abstractions

The bytecodes for a method implementation is represented as a list of abstract bytecode instructions (of type `Bytecode.jvm_instr`), corresponding to an abstract view of the JVM instruction set. For example, the `jvm_instr` data type includes the pseudo-instruction `Jlabel` which has no counterpart in the actual instruction set of the JVM.

In the SML-JVM toolkit, labels are used for specifying target(s) of branching instructions, and for specifying scope and entry point of exception handlers. When the abstract bytecode instruction sequence is emitted the labels therein are converted to addresses in the resulting binary code (cf. Section 5).

Moreover, the `jvm_instr` data type contains a number of amalgamated JVM instructions. For example, the abstract instruction `Jiconst` represents all JVM instructions pushing a constant `int` value onto the operand stack. When the abstract bytecode sequence is emitted, the appropriate JVM instruction is chosen. The translation performed by the bytecode emitter is:

Abstract instruction	JVM instruction
Jconst	ldc or ldc_w
Jaload	aload_<n>, aload, or wide aload
Jastore	astore_<n>, astore, or wide astore
Jdconst	dconst_<n> or ldc2_w
Jdload	dload_<n>, dload, or wide dload
Jdstore	dstore_<n>, dstore, or wide dstore
Jfconst	fconst_<f>, ldc, or ldc_w
Jfload	fload_<n>, fload, or wide fload
Jfstore	fstore_<n>, fstore, or wide fstore
Jgoto	goto or goto_w
Jiconst	iconst_<i>, bipush, sipush, ldc, or ldc_w
Jiinc	iinc or wide iinc
Jiload	iload_<n>, iload, or wide iload
Jistore	istore_<n>, istore, or wide istore
Jjsr	jsr or jsr_w
Jlconst	lconst_<l> or ldc2_w
Jlload	lload_<n>, lload, or wide lload
Jlstore	lstore_<n>, lstore, or wide lstore
Jnewarray	newarray, anewarray, or multianewarray
Jret	ret or wide ret
Jreturn	areturn, dreturn, freturn, ireturn, lreturn, or return

The bytecode emitter translates each of the above abstract bytecode instructions into the most compact of the corresponding JVM bytecode instructions.

By introducing this level of abstraction over the instruction set of the JVM, the user of the SML-JVM toolkit is relieved from worrying about details in the JVM instruction set, e.g. which of the JVM instructions `iconst_<i>`, `bipush`, `sipush`, and `ldc` should be used for pushing an integer constant onto the operand stack. The user just inserts an abstract `Jiconst` instruction with an `Int32.int` argument, and the bytecode emitter then chooses the most compact of the corresponding JVM instructions based on the value of the immediate integer argument.

All other abstract bytecode instructions than those listed above are mapped directly to the corresponding JVM instructions. For example, the abstract instruction `Jdup` simply maps to the JVM instruction `dup` (represented in binary form as opcode 89).

4 Generating A Class File

Generation of a physical Java class file from an abstract class/interface declaration is implemented in the function `Classfile.emit`. This function takes as arguments a ‘writer’ function, an initial constant pool, and a class declaration. The class declaration is then processed in two steps:

1. the abstract `class_decl` is converted to a more concrete `class_file` record,
2. the `class_file` record is output, e.g. to a physical file, using the ‘writer’ function for writing each single byte of output to the file.

A `class_file` record represents a class/interface declaration in binary format:

- abstract bytecode instructions for methods implemented in the class have been ‘emitted’ to binary form (cf. Section 5),
- constants and references to fields, methods and classes have been inserted into the constant pool (cf. Section 5.1),
- labels have been resolved to instruction addresses (cf. Section 5.2), and
- access flags for the class/interface and its members have been converted to binary representation.

The reason for generating a class file in two steps is that the constant pool for the resulting class file must be built before the file is actually written. The constant pool goes at the beginning of a binary class file, and its size is not known until it has been completed, i.e., until all constants and references have been inserted. Hence, the constant pool must be completed before the remaining parts of the class file can be output.

The constant pool argument to `Classfile.emit` is used as the basis of the constant pool of the resulting class file. This way, data referenced by non-standard attributes of the class/interface declaration can be included in the resulting class file. Such attributes would include, as part of their binary content, the constant pool indices of the referenced data. In case a class or interface uses only standard attributes, an empty constant pool should be passed as argument to `Classfile.emit`.

5 Code Emission

The conversion from abstract bytecode to binary representation of JVM bytecode instructions is implemented in the function `Emitcode.emit`. This function translates the abstract bytecode for a method to a byte vector holding the binary representation of the corresponding JVM bytecode instructions.

The translation involves mapping the abstract bytecode instructions to the actual JVM instructions (cf. Section 3), and for each abstract bytecode instruction choosing the most compact JVM instruction:

- For each `Jsconst` abstract instruction, the argument (a string literal) is inserted into the constant pool, and one of the JVM instructions `ldc` and `ldc_w` is emitted, with the resulting constant pool index as argument. If the constant pool index is less than 256 then `ldc` is used, otherwise `ldc_w` is used.
- For each `Jaload`, `Jdload`, `Jfload`, `Jiload`, and `Jlload` abstract instruction, if the index of the local variable being accessed is between 0 and 3, inclusive, then the corresponding JVM load instruction with an implicit immediate operand is emitted. Otherwise, if the index is between 4 and 255, inclusive, then the corresponding JVM load instruction with an immediate operand is emitted. Otherwise the corresponding `wide` JVM load instruction is emitted.
- For each `Jastore`, `Jdstore`, `Jfstore`, `Jistore`, and `Jlstore` abstract instruction, if the index of the local variable being accessed is between 0 and 3, inclusive, then the corresponding JVM store instruction with an implicit immediate operand is emitted. Otherwise, if the index is between 4 and 255, inclusive, then the corresponding JVM store instruction with an immediate operand is emitted. Otherwise the corresponding `wide` JVM store instruction is emitted.

- For each **Jdconst** abstract instruction, if the argument is 0.0 or 1.0 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and the JVM instruction **ldc2_w** is emitted with the resulting constant pool index as argument.
- For each **Jfconst** abstract instruction, if the argument is 0.0, 1.0, or 2.0 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and one of the JVM instructions **ldc** and **ldc_w** is emitted, depending on the resulting constant pool index.
- For each **Jgoto** abstract instruction, if the argument label has been resolved to an address, and if the instruction offset for that address is between -32768 and 32767 , inclusive, then the JVM instruction **goto** is emitted. Otherwise the JVM instruction **goto_w** is emitted. See also Section 5.2.
- For each **Jiconst** abstract instruction, if the argument is between -1 and 5 , inclusive, then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise, if the argument is between -128 and 127 , inclusive, then the JVM instruction **bipush** is emitted. Otherwise, if the argument is between -32768 and 32767 , inclusive, then the JVM instruction **sipush** is emitted. Otherwise the argument is inserted into the constant pool, and one of the JVM instructions **ldc** and **ldc_w** is emitted, depending on the resulting constant pool index.
- For each **Jiinc** abstract instruction, if the index of the local variable being accessed is between 0 and 255 , inclusive, and if the increment argument is between -128 and 127 , inclusive, then the JVM **iinc** instruction is emitted. Otherwise, the JVM instruction **wide iinc** is emitted.
- For each **Jret** abstract instruction, if the index of the local variable to be loaded is between 0 and 255 , inclusive, then the JVM instruction **ret** is emitted. Otherwise the JVM instruction **wide ret** is emitted.
- For each **Jjsr** abstract instruction, if the argument label has been resolved to an address, and if the offset is between -32768 and 32767 , inclusive, then the JVM instruction **jsr** is emitted. Otherwise the JVM instruction **jsr_w** is emitted. See also Section 5.2.
- For each **Jlconst** abstract instruction, if the argument is 0 or 1 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and the JVM instruction **ldc2_w** is emitted with the resulting constant pool index as argument.
- For each **Jnewarray** abstract instruction, if more than one dimension of an array is to be created at once then the JVM instruction **multianewarray** is emitted. Otherwise one of the JVM instructions **newarray** and **anewarray** is emitted, depending on the component type of the array (if the component type is a simple type **newarray** is used, otherwise **anewarray** is used).
- For each **Jreturn** abstract instruction, the JVM method return instruction corresponding to the return type of the method is emitted¹.

The binary representation of the emitted JVM instructions and their arguments is collected in a byte array (`Word8Array.array`). This array is created with the maximum bytecode size

¹The six different JVM method return instructions **areturn**, **dreturn**, **freturn**, **ireturn**, **lreturn**, and **return** are examples of redundancy in the JVM instruction set. These ‘typed’ instructions could readily be merged into one ‘untyped’ return instruction, and the JVM would then return a value of the proper type based on the return type of the method.

for a method: 65,535 bytes. When all abstract bytecode instructions for the method have been translated, a vector of JVM bytecodes is extracted from the byte array, corresponding to the part of the array that has actually been used. This vector of JVM bytecodes is output as method code when the class declaration is written to a class file (once bytecode has been generated for all methods of the class).

5.1 Building the Constant Pool

The `Constpool` module implements the `pool` data type and related operations. The implementation uses a hash table to achieve maximum sharing: when a constant or reference is inserted into the pool, it is first checked whether an identical value is already in the pool. If so, the index of that entry is returned; otherwise the value is inserted into the constant pool, and the fresh index of the new entry is returned.

In the process of translating the sequence of abstract bytecode instructions to a byte vector, the bytecode emitter builds the constant pool for the target class. As mentioned, emission of an `Jsconst`, `Jdconst`, `Jfconst`, `Jiconst`, or `Jlconst` abstract instruction may lead to a constant being inserted into the constant pool. In that case, one of the JVM instructions loading a constant pool entry is emitted, with the index of the constant pool entry as argument.

Similarly, all symbolic references to classes, fields and methods are inserted into the constant pool when the referencing instruction is emitted, and the index of the resulting constant pool entry is emitted as an argument to the instruction.

5.2 Label Resolution

The bytecode emitter also implements resolution of abstract bytecode labels to addresses in the target JVM bytecode. This is done via a hash table representing a map from a label to its ‘target status’, which is one of

`PENDING(ref phs)` meaning that a list `phs` of references to the label has been encountered, but the label has not yet been resolved to an address
`RESOLVED addr` meaning that the label has been resolved to the address `addr`.

When a reference to a label is encountered, if the label has already been resolved to an address, then the target JVM branching instruction is emitted with that address as argument. Otherwise the target JVM branching instruction is emitted with a dummy argument (placeholder), and a reference to the placeholder is added to the list of `PENDING` references to the label (if any).

When a `Jlabel` abstract bytecode instruction is encountered, if there are any `PENDING` references to the label, these are backpatched to the address of the following target JVM instruction. The label is then marked as `RESOLVED` in the label map. No target JVM bytecode is emitted for the `Jlabel` instruction itself.

The bytecode emitter is more conservative than strictly necessary with respect to the abstract instructions `Jgoto` and `Jjsr`: only in case the branch target lies before the branch instruction will the most compact of the JVM instructions `goto` and `goto_w` or `jsr` and `jsr_w` be chosen. When the branch target lies after the branching instruction itself the JVM instruction `goto_w` or `jsr_w` will always be used. This is because bytecode emission is implemented as a single pass over the abstract bytecode sequence. Using a more advanced multi-pass algorithm would make it possible to always choose the most compact JVM branching instruction.

6 Auxiliary Modules

In order to support the full range of constants defined in the Java class file format[3], 16-bit, 32-bit and 64-bit integer and floating-point constants are used in a number of places in the SML-JVM toolkit. An example of this was shown in the field declaration example of Section 2 where the integer argument to the `Cint` constructor is of type `Int32.int`.

Moscow ML provides only one `Int` structure (31-bit signed integers), one `Word` structure (31-bit unsigned integers), and one `Real` structure (64-bit floating-point values). Hence, a set of utility modules has been implemented:

<code>Int32</code>	32-bit signed integers
<code>Int64</code>	64-bit signed integers
<code>Real32</code>	32-bit floating-point values
<code>Real64</code>	64-bit floating-point values
<code>Word16</code>	16-bit unsigned integers
<code>Word32</code>	32-bit unsigned integers
<code>Word64</code>	64-bit unsigned integers

These modules provide a limited subset of the corresponding SML Basis Library modules[2], primarily conversion from the usual integer and real types (`Int.int`, `Word.word`, and `Real.real`). Additionally, functions for conversion to byte vectors are provided.

The auxiliary modules are used in the conversion from abstract class declarations to concrete `class_file` records (implemented in `Classfile.emit`).

7 Downloading

The SML-JVM toolkit is available for anonymous FTP from:

```
ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/sml-jvm-toolkit.zip
```

The toolkit was designed and implemented by Peter Bertelsen (<http://www.dina.kvl.dk/~pmb>).

References

- [1] J. Gosling et al. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0-201-63451-1.
- [2] J. Reppy et al. Standard ML Basis Library. Technical report, Bell Labs, Lucent Technologies, 1997. Available at <http://www.cs.bell-labs.com/~jhr/sml/basis>.
- [3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996. ISBN 0-201-63452-X.
- [4] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [5] R. Milner, M. Tofte, R. Harper, and D.B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.