Translating Java bytecode to Simple

Semester Thesis Report

Scheidegger Roman

June 2010

Chair of Programming Methodology
http://www.pm.inf.ethz.ch/
Department of Computer Science ETH Zurich

Contents

1. Introduction

The Java bytecode language is composed by more than 200 statements. Intuitively, it is an intermediate language between source and machine code. Thus the most part of statements is specific for a given type(e.g. sum operators of integers, floats, etc.) as they are intended to optimize the execution (e.g. loading from and storing to the first local variable, the second one, etc.). In addition, Java bytecode is not structured, i.e. it contains goto's and conditional jumps. Finally, values are passed through an operand stack. E.g. in order to add two integer numbers, they are pushed on the operand stack, and a sum operator adds them pushing on the top of the stack the resulting value.

1.1 Goal

The goal of this semester project was to develop and implement a translation from Java bytecode programs to a particular language called Simple, composed by a small set of statements (i.e. assignments, variable declarations, throws of exceptions, method calls, and field accesses), and to represent programs through control flow graphs. This language is used by Sample (Static Analyzer for Multiple Programming LanguagEs), a new generic static analyzer developed by Pietro Ferrara at ETH, and had already been developed and implemented together with the control flow graph structure. A translation from Scala code to Simple was developed as well. My assignment was to translate Java bytecode programs into Simple using the control flow graph structure. The main issues were to

- abstract away the operand stack
- build up the control flow graph starting from unstructured code
- inject the information contained in the bytecode (e.g. type information) into its simplified representation

2. Simple

In this chapter a short overview of Simple is given. To have a more comprehensive insight about it have a look at [1].

2.1 Statements

We'd like to translate Java bytecode to Simple, a very compact language consisting of essentially 8 main statements:

- Variable(ID) abbreviated with V(..)
  Example: `x` represented by Variable('x')

- NumericalConstant(Value, Type) abbreviated with NC(..)
  Example: `1` represented by NumericalConstant(1,I)

- VariableDeclaration(Variable, Type, Right) abbreviated with VD(..)
  Example: `int x = 1` represented by
  VariableDeclaration(Variable('x'), I, NumericalConstant(1,I))

- FieldAccess(Object/s, Field, Type) abbreviated with FA(..)
  Example: `x.y` represented by FieldAccess(Variable('x'), y, I)

- MethodCall(Method, Parameters, Returntype) abbreviated with MC(..)
  Example: `x.+(z)` represented by
  MethodCall(FieldAccess(Variable('x'), '+', I), < Variable('z') > , I)

- Assignment(Left, Right) abbreviated with A(..)
  Example: `x = y` represented by Assignment(Variable('x'), Variable('y'))

- New(Type) abbreviated with N(..)
  Example: `new IOException()` represented by New('IOException')

- Throw(Expression) abbreviated with T(..)
  Example: `throw new IOException` represented by Throw(New('IOException'))

Note that the Java bytecode descriptor notation was used here, so I stands for an integer type. More information about this notation can be found in the VM Spec[2] chapter 4.3. Also a simplified notation was used here do describe the statements, not taking into account parameters which are irrelevant for the Java bytecode translation. < ... > was used to denote lists.

For some languages it's common to distinguish between expressions and statements in a way that expressions return values (e.g. `3 + variable`) while statements do not (e.g. `x = y + x`). In some cases a statement is even also considered as an expression(e.g. a method call that returns a value). To preserve simplicity and expresiveness of Simple an approach was chosen where basically everything is an expression and a statement is just a special case of an expression, returning an empty value(`Unit`).

Another thing that could lead to a more complex language are arithmetic expressions or things like type castings, that are natively defined by some programming languages. Again a generalized approach was chosen for Simple where everything is an object. With this assumption also numerical constants are objects and for example arithmetic operations like `3 + 2` can be represented in a much simpler and generalized way like `3.+(2)` to overcome the need of natively defined concepts.

2.1.1 ProgramPoint

To retain a certain correspondence between Simple and the language translated to it, in our case Java bytecode, each statement carries a ProgramPoint object giving us a location in the original program source. This can be of great use because it enables the analyzer to go back from a problem recognized in Simple to the location in the original program code.

2.1.2 Type

In a lot of the statements we do specify types (e.g. the return type of a method call). We do this by using a class Type to represent the type hierarchy. It provides some methods (i.e isObject(), isNumericalType(), isStatic(),

getName() etc.) to extract some information about the given type or type hierarchy.

2.2 Control Flow Graph

A CFG is basically a weighted graph built up for each method to capture the structure of possible execution paths in the code, where nodes are list of statements. Each node can then either have one outgoing edge with no weight attached to it to represent an unconditional jump, or have two outgoing edges with boolean weights true and false to represent a conditional jump. For the latter case we need to put a condition like `x.>=(0)` as the last statement to the node which can be evaluated to a boolean value. So the edges are pointing to the nodes that are executed if the condition is evaluated to the corresponding value.

2.3 ClassDefinition

In order to represent the structure of a class there are three major components:

- FieldDeclaration(Modifiers, Name, Type, Right) abbreviated with FD(..)
  Example: `private int x` represented by
  FieldDeclaration(<private>, 'x', I, null)

- MethodDeclaration(Modifiers, Name, Parameterictypes,
  Arguments, Returntype, Body, Precondition, Postcondition, Invariant) abbreviated with MD(..)
  Example: `public static int evaluate(int z)` represented by
  MethodDeclaration(<public,static>, 'evaluate', null, <VD('z',I,null)>, I, cfg, null, null, null)

- ClassDefinition(Modifiers, Name, Parametrictypes, Extends, Fields, Methods, Package) abbreviated with CD(..)
  Example: `public class A extends B {}` represented by
  ClassDefinition(<public>, 'A', null, 'B', null, null, null)

For each method of a class the CFG of the code is built up and used to generate a MethodDeclaration with additional information like the access modifiers of the method(e.g `public`), the argument count and argument types,

the return type and of course also the name of the method. Analagously FieldDeclarations are built for each field of the class and then all Field- and MethodDeclarations are added to a ClassDefinition which contains general information of the class like package name, super class, access modifiers and the class name itself. This ClassDefinition wraps all the needed information together and is all we need to pass to the analyzer for the analysis of the class. So our end goal will be to translate a given Java bytecode class file to a coresponding ClassDefinition.

3. JavaToSimple

The main goal of this project was to transform Java bytecode to Simple as described in section 2. Therefore a library called JavaToSimple was built to achieve this transformation. Here is an overview of the classes provided:

ClassFileParser	Main parser
LocalVariableType	Helper class to keep track on actual variable declaration status
TypedStatement	Combination of a statement with it's type information
TypeExtractor	Provides some helpful functions to extract type and access modifier information from Java bytecode type descriptors and access flags
SimpleClasses.JavaClassIdentifier	Identifier for a Java class
SimpleClasses.JavaMethodIdentifier	Identifier for a Java method
SimpleClasses.JavaPackageIdentifier	Identifier for a Java package
SimpleClasses.JavaProgramPoint	Unique program point in the Java bytecode
SimpleClasses.JavaType	Stores type information

With this library you can obtain the ClassDefinition of a Java classfile needed for Sample in a very simple way:

```
ClassDefinition cd = new ClassFileParser(name).getClassDefinition();
```

3.1 ClassFileParser

The ClassFileParser's constructor just takes the path of a Java classfile as an argument to load and parse it and in the end generate the complete ClassDefinition, which then later can be retrieved via the function getClassDefinition(). This ClassDefinition can be analyzed by Sample.

When parsing the classfile first general properties of the class like access modifiers, superclass and the package name are extracted. Afterwards all the FieldDeclarations with their access modifiers are read out and then the MethodDeclarations are generated. In the end all this information is put together to a ClassDefinition.

3.1.1 Generation of MethodDeclarations

When generating the MethodDeclarations we do also read out the access modifiers first. For non-abstract and non-native methods then the given bytecode is translated to a CFG composed by Simple statements and that's actually the core contribution of this project.

Translating the opcodes

Because of the unstructured nature of the bytecode the code is parsed sequentally, each bytecode instruction one after the other and translated by mimicing the behavior of the JVM [2]. To load and read the Java classfiles the Javassist [3] library was used. This enabled the easy retrieval of bytecode instructions and also certain other useful information like types, exception handler positions etc.

It is obvious that there isn't a one to one mapping between Java bytecode and Simple so generally a sequence of bytecode instructions is mapped to a Simple statement, possibly taking as arguments other Simple statements. To actually build this complex nested structure out of the sequential structure of bytecode, a virtual operand stack was used to mimic the behaviour of the JVM. If we are translating an opcode that would put an intermediate result on the operand stack in the JVM we build an intermediate Simple statement and push this to the virtual operand stack. Opcodes that retrieve arguments from the operand stack in the JVM are then translated by retrieving their operands from the virtual operand stack. Opposite to this kind of instructions that builds statements and pushes the result to the virtual operand stack we have instructions that build a statement which is final and needs no more processing. Those opcodes are called *flush-causing* and in their case we call the __flushids() function whichs adds the given statement to the list of parsed statements. Each parsed statement obtains a unique id and we have two maps called instostaid and staidtoins to remember which bytecode address was mapped to which statement id and vice-versa. All this information is stored when calling __flushids(). Last there are some special bytecode instructions like `goto` or `ifeq` which cause jumps in the bytecode. We need to remember this jump information, because we need it later to build the

CFG. So in the case of unconditional jumps like for `goto` we have a map called unconditionaljumps where we put a jump in the sense of a <source, target>-tuple. For conditional jumps there's always a condition evaluating to a boolean value and we always need to add two jumps, one for the true case and one for the false case. Thus we have two additional maps called truejumps and falsejumps where we put the corresponding jumps in the case of a conditional jump. Note that the source and target addresses stored for each jump are still bytecode addresses.

This is best illustrated with an example. The simple Java program:

```
int b = 3;
int a = 5 + b;
```

is first translated to Java bytecode like this

```
1: iconst_3
2: istore_2
3: iconst_5
4: iload_2
5: iadd
6: istore_3
```

and this is then represented in Simple - actually a simplified version without types is used here to make it more readable - in the following form:

```
1: A(V('v2'), NC(3));
2: A(V('v3'), MC(FA(V('v2'), '+'), NC(5)));
```

So lets have a closer look at how we get from Java bytecode to Simple statements.

First `iconst_3` is parsed so we create a NumericalConstant with value three and push it onto the virtual operand stack. Next instruction is `istore_2`, which in the JVM would pop the value at the operand stack and store it in the local variable number 2. So when translating this instruction we pop the top statement from our virtual operand stack and create an Assignment statement with a newly created Variable statement referencing the second local variable as the left hand side and the popped statement as the right hand side. So we see that one bytecode instruction can generate more than one statement in one step. The `istore_2` instruction is actually a *flush-causing* one and therefore we call __flushids() with the newly built Assignment statement as a parameter.

It is important to note that either the built statements are put to the list of parsed statements or to the virtual operand stack depending on wheather we have a flush-causing instruction or not. So generally a indicator to tell if a bytecode instruction is flush-causing or not is the behaviour of it in the JVM. If the instruction would not put something to the operand stack it is a flush-causing instruction in most cases and vice-versa.

So to continue with the above example. We then have again an `iconst` instruction so a NumericalConstant is pushed on the virtual operand stacked, followed by an `iload_2` instruction which actually generates a Variable statement referencing again the second local variable and puts this on to the virtual operand stack. The `iadd` operation then pops the two top statements and generates a MethodCall to the '+'-method of the first object and then pushes this MethodCall again to the virtual operand stack, like the JVM would push the result to the operand stack. Actually in this step we put a MethodCall on a FieldAccess to the operand stack. So note that actually also a single bytecode instruction can cause creation of more than one Simple statement. Then we have again an `istore`, so again an Assignment is generated and flushed.

So this is the general translation process of bytecode instructions giving us in the end a list of parsed Simple statements in the order they were created, two maps storing the correspondence between bytecode addresses and statement ids and three maps holding the information about the jumps present in the code. You can have a look at section 3.4 to have a detailed overview of how which Java bytecode instruction is mapped to what Simple statements.

Building Nodes

As soon as we have parsed all the bytecode instructions to Simple statements we need to build the CFG out of them. We do this by first building multiple code blocks out of the parsed statements and then adding edges inbetween them. These code blocks should be as large as possible and created in such a way that the source of a jump is always at the end of a block and the target of a jump is always at the beginning of a block, because it is not possible later for an edge to point to somewhere in the middle of a block. An edge is always outgoing from the end of a block to the begin of a block. Given a list of Simple statements in the order they were parsed and some maps giving us the jumps in the bytecode we want to build the CFG so that the different code blocks of it are as large as possible. We achieve this by simply going through the list of parsed statements and adding the statements to the current block until we reach a reason to begin a new block. Reasons to begin

a new block are:

- current statement is the source of an outgoing jump
- current statement is the target of a jump

So for the first case we add the current statement to the current block and then begin a new block. In the second case we immediately begin a new block and add the current statement already to the new block. A statement may possibly be both the source and the target of a jump. In this case a block with just this statement should be created. It's also important to note that for the second split reason in some cases we also need to add an additional edge between the last and the newly generated block. This is because in 'normal' program flow the first statement of the new block would be following the last statement of the old block. So we need to check wheather this jump is already existing and if not we need to add it manualy to retain the original structure.

Also for this process it is important to remember which statement was added to which block and so we store those mappings again in two maps.

Adding Edges

After building the blocks we want to add the edges to the CFG, but they are actually given in bytecode indexes. So with all our maps we can determine which bytecode address, was mapped to which statement and for the statement we know to which block it was added. Because of they way we splitted our codeblocks we know that if a bytecode address is the source of a jump, the corresponding statement is always at the end of a code block and if a bytecode address is the target of a jump the corresponding statement is always at the beginning of a code block. Thus we need just to retrieve the code blocks of the two bytecode addresses making up the jump and then add a corresponding edge inbetween the blocks. The weight of the edge is basically given by the map we found the jump in(e.g. true for a jump in truemaps).

Parsing jsr calls

Java bytecode subroutines is a language feature not available in Simple. To handle this, all the subroutine calls were treated with code inlining. As soon as the parsing process arrives at a `jsr` call, the return address as also the destination address is pushed to a jsr-stack. Then the sequential parsing proceeds at the destination given by the jsr call. As soon as we arrive at

a `ret` instruction we pop the return address from the jsr-stack and parsing continues at the next instruction after the return address. We are using a stack here to make it possible to parse nested jsr calls.

Because we have inserted instructions by inlining we need to give them unique adresses, so that also jumps in a subroutine are recognized as beeing in a subroutine and resolved in the right way. This is resolved by using a helper function called __getactualindex which takes an integer index as an argument and returns a string address. This function basically returns the index as a string if we are in ordinary parsing mode, but as soon as the jsr-stack is non-empty, indicating that we are currently parsing a jsr-call, a string like `JSRfrom:to` is appended to the index giving those instructions a unique id, depending which subroutine was called from which location.

By consequently using the __getactualindex function everywhere where we need to reference some bytecode indexes, we achieve a consistent addressing schema.

3.2 LocalVariableType

It is possible for Java bytecode to use a local variable multiple times with different types and because Simple doesn't support this we need to keep track of such redeclarations. We treat such redeclarations in Simple like new variables with new type and name. The LocalVariableType class is used to remember the current type and revision of a local variable. So we can always use this information to check wheather we have the same variable as before, or a new one.

We know in advance how many local variables are used so we have a list of LocalVariableType's. As local variables are also used to pass arguments, first the local variable type of such local variables are set to the argument types. Then each time we have an assignment to a local variable there is a check to ensure that the types are the same. If not, we set the new type and internally the variable revision is increased. VariableIdentifiers for Simple are then always composed out of the number and the revision to ensure that we have a 'new' variable if the revision increased.

3.3 JavaProgramPoint

This class represents a point in Java bytecode to use as described in section 2.1.1. It basically takes three parameters, namely two strings called *classname* and *block* and an integer called *row*. The classname parameter is

trivial and always corresponds to the name of the class in which we want to describe a program point. The block parameter by default is a method name of the given class and in such a case the row parameter corresponds to the byte offset in the code attribute of the given method. The other cases of the block parameter are as follows:

- `::classfile` This is used for the program points of the ClassDefinition and in this case the row parameter is set to zero.
- `::field_info` This is used for porgram points of FieldDeclaration's and in that case the row parameter gives us the number of the declared field.
- `::method_info` This is used for program points of MethodDeclaration's and in that case the row parameter gives the number of the declared method.

3.4 Translation Details

In this chapter we give a detailed overview of how each Java bytecode instruction is translated. Again the Java bytecode descriptor notation is used here to indicate types. If nothing is indicated the given Simple statements are put to the virtual operand stack. If the statement is added to the list of parsed statements instead we indicated this with the term *flush-causing*.
Abbreviations used:

- OCV Opcode Value: The value is determined by the instruction itself.
- BAV Byte Argument Value: The value is given as a sequence of bytes after the opcode.
- CPV Constant Pool Value: The value is given by a reference to an item in the constant pool.
- VI Variable Identifier
- VT Variable Type
- tempN Name of temporary Variable
- OSAx Operand Stack Argument x: The value ist taken from the position x starting from the top of the virtual operand stack.
- `<..>|` Indicates a list.

`nop(0)`	No Statement built
`aconst_null(1)`	NC('null','Ljava/lang/Object;')
`iconst_m1(2)`, `iconst_0(3)`, `iconst_1(4)`, `iconst_2(5)`, `iconst_3(6)`, `iconst_4(7)`, `iconst_5(8)`	NC(OCV, I) Helper function: `__pushconstant`
`lconst_0(9)`, `lconst_1(10)`	NC(OCV, J) Helper function: `__pushconstant`
`fconst_0(11)`, `fconst_1(12)`, `fconst_2(13)`	NC(OCV, F) Helper function: `__pushconstant`
`dconst_0(14)`, `dconst_1(15)`	NC(OCV, D) Helper function: `__pushconstant`
`bipush(16)`, `sipush(17)`	NC(BAV, I) Helper function: `__pushconstant`
`ldc(18)`, `ldc_w(19)`, `ldc2_w(20)`	NC(CPV, CPV) Helper function: `__pushconstant`
`iload(21)`, `lload(22)`, `fload(23)`, `dload(24)`, `aload(25)`	Variable(VI) Helper function: `__pushvariable` Note: The VariableIdentifier is built with the variable number given as a bytecode argument and the current revision of the variable, tracked with `localtypes`.

`iload_0(26)`, `iload_1(27)`, `iload_2(28)`, `iload_3(29)`, `lload_0(30)`, `lload_1(31)`, `lload_2(32)`, `lload_3(33)`, `fload_0(34)`, `fload_1(35)`, `fload_2(36)`, `fload_3(37)` `dload_0(38)`, `dload_1(39)`, `dload_2(40)`, `dload_3(41)`, `aload_0(42)`, `aload_1(43)`, `aload_2(44)`, `aload_3(45)`	Variable(VI) Helper function: `__pushvariable` Note: The VariableIdentifier is built with the variable number given as an instruction suffix and the current revision of the variable, tracked with `localtypes`.
`iaload(46)`	MC(FA(OSA2, 'apply', (I)I), <OSA1>, I) Helper function: `__pusharrayaccessread`
`laload(47)`	MC(FA(OSA2, 'apply', (I)J), <OSA1>, J) Helper function: `__pusharrayaccessread`
`faload(48)`	MC(FA(OSA2, 'apply', (I)F), <OSA1>, F) Helper function: `__pusharrayaccessread`
`daload(49)`	MC(FA(OSA2, 'apply', (I)D), <OSA1>, D) Helper function: `__pusharrayaccessread`
`aaload(50)`	MC(FA(OSA2, 'apply', (I)Ljava/lang/Object;), <OSA1>, Ljava/lang/Object;) Helper function: `__pusharrayaccessread`
`baload(51)`	MC(FA(OSA2, 'apply', (I)I), <OSA1>, I) Helper function: `__pusharrayaccessread`

`caload(52)`	MC(FA(OSA2, 'apply', (I)I), <OSA1>, I) Helper function: `__pusharrayaccessread`
`saload(53)`	MC(FA(OSA2, 'apply', (I)I), <OSA1>, I) Helper function: `__pusharrayaccessread`
`istore(54)`, `lstore(55)`, `fstore(56)`, `dstore(57)`, `astore(58)`	A(V(VI),OSA1) or VD(V(VI), VT, OSA1) Helper function: `__variableassignment` flush-causing Note: The VariableIdentifier is built with the variable number given as a bytecode argument and the current revision of the Variable, tracked with `localtypes`. `localtypes` is also used to determine wheather we have a re-declaration or an assignment. The VariableType is determined from the operand stack argument OSA1.

`istore_0(59)`, `istore_1(60)`, `istore_2(61)`, `istore_3(62)`, `lstore_0(63)`, `lstore_1(64)`, `lstore_2(65)`, `lstore_3(66)`, `fstore_0(67)`, `fstore_1(68)`, `fstore_2(69)`, `fstore_3(70)`, `dstore_0(71)`, `dstore_1(72)`, `dstore_2(73)`, `dstore_3(74)`, `astore_0(75)`, `astore_1(76)`, `astore_2(77)`, `astore_3(78)`	A(V(VI),OSA1) or VD(V(VI), VT, OSA1) Helper function: `__variableassignment` flush-causing Note: The VariableIdentifier is built with the variable number given as an instruction suffix and the current revision of the Variable, tracked with `localtypes`. `localtypes` is also used to determine wheather we have a re-declaration or an assignment. The VariableType is determined from the operand stack argument OSA1.
`iastore(79)`	MC(FA(OSA3, 'update', (II)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`lastore(80)`	MC(FA(OSA3, 'update', (IJ)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`fastore(81)`	MC(FA(OSA3, 'update', (IF)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing

`dastore(82)`	MC(FA(OSA3, 'update', (ID)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`aastore(83)`	MC(FA(OSA3, 'update', (ILjava/lang/Object;)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`bastore(84)`	MC(FA(OSA3, 'update', (IZ)V), <OSA2, OSA1>, V) or MC(FA(OSA3, 'update', (IB)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing Note: The used type is determined with the parameter OSA1 on the operand stack.
`castore(85)`	MC(FA(OSA3, 'update', (IC)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`sastore(86)`	MC(FA(OSA3, 'update', (IS)V), <OSA2, OSA1>, V) Helper function: `__pusharrayaccesswrite` flush-causing
`pop(87)`	No statement is created. The statement on top of the virtual operand stack is popped and flushed. flush-causing

`pop2(88)`	No statement is created. The statement on top of the virtual operand stack is popped and flushed. If this statement is not of type double or long a second statement is popped and flushed. flush-causing
`dup(89)`	No statement is created. The statement on top of the virtual operand stack is popped and then put on the operand stack twice.
`dup_x1(90)`	No statement is created. The two top statements(OSA1, OSA2) of the virtual operand stack are popped and then put again to the virtual operand stack in the following order: OSA1, OSA2, OSA1.
`dup_x2(91)`	No statement is created. The two top statements(OSA1, OSA2) of the virtual operand stack are popped. If OSA2 is either of type double or long the statements are put back on the operand stack in the following order: OSA1, OSA2, OSA1. Otherwise a third statement(OSA3) is popped and then these statements are put back to the virtual operand stack in the following order: OSA1, OSA3, OSA2, OSA1
`dup2(92)`	No statement is created. The top statement(OSA1) of the virtual operand stack is popped. If OSA1 is either of type double or long the statements is put back on the operand stack twice. Otherwise a second statement(OSA2) is popped and then these statements are put back to the virtual operand stack in the following order: OSA2, OSA1, OSA2, OSA1
`dup2_x1(93)`	No statement is created. The two top statements(OSA1, OSA2) of the virtual operand stack are popped. If OS1 is either of type double or long then the statements are put back again to the virtual operand stack in the following order: OSA1, OSA2, OSA1. Otherwise a third statement(OS3) is popped first and then all three statements are pushed back in the following order: OSA2, OSA1, OSA3, OSA2, OSA1

`dup2_x2(94)`	No statement is created. The two top statements(OSA1, OSA2) of the virtual operand stack are popped. If OSA1 and OSA2 are both either of type double or long then the statements are put back again to the virtual operand stack in the following order: OSA1, OSA2, OSA1. If only OSA1 is either of type double or long, a third statement (OSA3) is popped and then all three statements are pushed back in the following order OSA1, OSA3, OSA2, OSA1. If neither OSA1 nor OSA2 is type long or double a third statement OSA3 is popped of the virtual operand stack. If OS3 is of type double or long, the three statements are pushed back in the following order: OSA2, OSA1, OSA3, OSA2, OSA1. Otherwise a fourth statement OSA4 is popped and the four statements are pushed back like this: OSA2, OSA1, OSA4, OSA3, OSA2, OAS1
`swap(95)`	No statement is created. The two statements(OSA1, OSA2) on top of the virtual operand stack are popped and then pushed on the operand stack in swapped order: OSA1, OSA2.
`iadd(96)`	MC(FA(OSA2, '+', (I)I), <OSA1 >, I) Helper function: `__pushbioperator`
`ladd(97)`	MC(FA(OSA2, '+', (J)J), <OSA1 >, J) Helper function: `__pushbioperator`
`fadd(98)`	MC(FA(OSA2, '+", (F)F), <OSA1 >, F) Helper function: `__pushbioperator`
`dadd(99)`	MC(FA(OSA2, '+', (D)D), <OSA1 >, D) Helper function: `__pushbioperator`
`isub(100)`	MC(FA(OSA2, '-', (I)I), <OSA1 >, I) Helper function: `__pushbioperator`
`lsub(101)`	MC(FA(OSA2, '-', (J)J), <OSA1 >, J) Helper function: `__pushbioperator`

`fsub(102)`	MC(FA(OSA2, '-', (F)F), <OSA1 >, F) Helper function: `__pushbioperator`
`dsub(103)`	MC(FA(OSA2, '-', (D)D), <OSA1 >, D) Helper function: `__pushbioperator`
`imul(104)`	MC(FA(OSA2, '*', (I)I), <OSA1 >, I) Helper function: `__pushbioperator`
`lmul(105)`	MC(FA(OSA2, '*', (J)J), <OSA1 >, J) Helper function: `__pushbioperator`
`fmul(106)`	MC(FA(OSA2, '*', (F)F), <OSA1 >, F) Helper function: `__pushbioperator`
`dmul(107)`	MC(FA(OSA2, '*', (D)D), <OSA1 >, D) Helper function: `__pushbioperator`
`idiv(108)`	MC(FA(OSA2, '/', (I)I), <OSA1 >, I) Helper function: `__pushbioperator`
`ldiv(109)`	MC(FA(OSA2, '/', (J)J), <OSA1 >, J) Helper function: `__pushbioperator`
`fdiv(110)`	MC(FA(OSA2, '/', (F)F), <OSA1 >, F) Helper function: `__pushbioperator`
`ddiv(111)`	MC(FA(OSA2, '/', (D)D), <OSA1 >, D) Helper function: `__pushbioperator`
`irem(112)`	MC(FA(OSA2, '%', (I)I), <OSA1 >, I) Helper function: `__pushbioperator`
`lrem(113)`	MC(FA(OSA2, '%', (J)J), <OSA1 >, J) Helper function: `__pushbioperator`
`frem(114)`	MC(FA(OSA2, '%', (F)F), <OSA1 >, F) Helper function: `__pushbioperator`

drem(115)	MC(FA(OSA2, '%', (D)D), <OSA1 >, D) Helper function: __pushbioperator
ineg(116)	MC(FA(OSA1, 'negate', ()I), null, I) Helper function: __pushoperator
lneg(117)	MC(FA(OSA1, 'negate', ()J), null, J) Helper function: __pushoperator
fneg(118)	MC(FA(OSA1, 'negate', ()F), null, F) Helper function: __pushoperator
dneg(119)	MC(FA(OSA1, 'negate', ()D), null, D) Helper function: __pushoperator
ishl(120)	MC(FA(OSA2, '<<', (I)I), <OSA1 >, I) Helper function: __pushbioperator
lshl(121)	MC(FA(OSA2, '<<', (J)J), <OSA1 >, J) Helper function: __pushbioperator
ishr(122)	MC(FA(OSA2, '>>', (I)I), <OSA1 >, I) Helper function: __pushbioperator
lshl(123)	MC(FA(OSA2, '>>', (J)J), <OSA1 >, J) Helper function: __pushbioperator
iushr(124)	MC(FA(OSA2, '>>>', (I)I), <OSA1 >, I) Helper function: __pushbioperator
lushr(125)	MC(FA(OSA2, '>>>', (J)J), <OSA1 >, J) Helper function: __pushbioperator
iand(126)	MC(FA(OSA2, '&', (I)I), <OSA1 >, I) Helper function: __pushbioperator
land(127)	MC(FA(OSA2, '&', (J)J), <OSA1 >, J) Helper function: __pushbioperator

ior(128)	MC(FA(OSA2, '|', (I)I), <OSA1 >, I) Helper function: __pushbioperator
lor(129)	MC(FA(OSA2, '|', (J)J), <OSA1 >, J) Helper function: __pushbioperator
ixor(130)	MC(FA(OSA2, '^ ', (I)I), <OSA1 >, I) Helper function: __pushbioperator
lxor(131)	MC(FA(OSA2, '^ ', (J)J), <OSA1 >, J) Helper function: __pushbioperator
iinc(132)	MC(FA(V(VI), '+', (I)V), <NC(BAV, I) >, V) flush-causing
i2l(133)	MC(FA(OSA1, 'i2l', ()J), null, J) Helper function: __pushoperator
i2f(134)	MC(FA(OSA1, 'i2f', ()F), null, F) Helper function: __pushoperator
i2l(135)	MC(FA(OSA1, 'i2d', ()D), null, D) Helper function: __pushoperator
l2i(136)	MC(FA(OSA1, 'l2i', ()I), null, I) Helper function: __pushoperator
l2f(137)	MC(FA(OSA1, 'l2f', ()F), null, F) Helper function: __pushoperator
l2d(138)	MC(FA(OSA1, 'l2d', ()D), null, J) Helper function: __pushoperator
f2i(139)	MC(FA(OSA1, 'f2i', ()I), null, I) Helper function: __pushoperator
f2l(140)	MC(FA(OSA1, 'f2l', ()J), null, J) Helper function: __pushoperator

f2d(141)	MC(FA(OSA1, 'f2d', ()D), null, D) Helper function: __pushoperator
d2i(142)	MC(FA(OSA1, 'd2i', ()I), null, I) Helper function: __pushoperator
d2l(143)	MC(FA(OSA1, 'd2l', ()J), null, J) Helper function: __pushoperator
d2f(144)	MC(FA(OSA1, 'd2f', ()F), null, F) Helper function: __pushoperator
i2b(145)	MC(FA(OSA1, 'i2b', ()I), null, I) Helper function: __pushoperator
i2c(146)	MC(FA(OSA1, 'i2c', ()I), null, I) Helper function: __pushoperator
i2s(147)	MC(FA(OSA1, 'i2s', ()I), null, I) Helper function: __pushoperator
lcmp(148)	MC(FA(OSA2, 'compare', (J)I), <OSA1 >, I) Helper function: __pushbioperator
fcmpl(149)	MC(FA(OSA2, 'comparel', (F)I), <OSA1 >, I) Helper function: __pushbioperator
fcmpg(150)	MC(FA(OSA2, 'compareg', (F)I), <OSA1 >, I) Helper function: __pushbioperator
dcmpl(151)	MC(FA(OSA2, 'comparel', (D)I), <OSA1 >, I) Helper function: __pushbioperator
dcmpg(152)	MC(FA(OSA2, 'compareg', (D)I), <OSA1 >, I) Helper function: __pushbioperator
ifeq(153)	MC(FA(OSA1, '==', (I)Z), <NC(0, I) >, Z) Helper function: __comparisonzero

`ifne(154)`	MC(FA(OSA1, '!=', (I)Z), <NC(0, I) >, Z) Helper function: `__comparisonzero`
`iflt(155)`	MC(FA(OSA1, '<', (I)Z), <NC(0, I) >, Z) Helper function: `__comparisonzero`
`ifge(156)`	MC(FA(OSA1, '>=', (I)Z), <NC(0, I) >, Z) Helper function: `__comparisonzero`
`ifgt(157)`	MC(FA(OSA1, '>', (I)Z), <NC(0, I) >, Z) Helper function: `__comparisonzero`
`ifle(158)`	MC(FA(OSA1, '<=', (I)Z), <NC(0, I) >, Z) Helper function: `__comparisonzero`
. `if_icmpeq(159)`	MC(FA(OSA2, '==', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_icmpne(160)`	MC(FA(OSA2, '!=', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_icmplt(161)`	MC(FA(OSA2, '<', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_icmpge(162)`	MC(FA(OSA2, '>=', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_icmpgt(163)`	MC(FA(OSA2, '>', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_icmple(164)`	MC(FA(OSA2, '<=', (I)Z), <OSA1 >, Z) Helper function: `__comparison`
`if_acmpeq(165)`	MC(FA(OSA2, '==', (Ljava/lang/Object;)Z), <OSA1 >, Z) Helper function: `__comparison`

`if_acmpne(166)`	MC(FA(OSA2, '!=', (Ljava/lang/Object;)Z), <OSA1 >, Z) Helper function: `__comparison`
`goto(167)`	No statement is created. A jump from the current position to the given destination is added to the unconditionaljumps map.
`jsr(168)`	No statement is created. By putting a token of the form JSRsource:target on the jsrcalls stack the JSR parsing mode is enabled. Parsing continues at the target bytecode address.
`ret(169)`	No statement is created. The return address is fetched from the jsrcalls stack to determine the return address of the current JSR call and parsing continues at the given return address.
`tableswitch (170)`	Translates the tableswitch opcode to a series of `if(index == exp1) goto dest1` expressions. Each of the comparisons is added to the list of parsed statements so its actually flush-causing and corresponding jumps are added. Helper function: `__tableswitch` flush-causing
`lookupswitch (171)`	Translates the lookupswitch opcode to a series of `if(key == exp1) goto dest1` expressions. Each of the comparisons is added to the list of parsed statements so its actually flush-causing and corresponding jumps are added. Helper function: `__lookupswitch` flush-causing
`ireturn(172)`, `lreturn(173)`, `freturn(174)`, `dreturn(175)`, `areturn(176)`	The statement on the virtual operand stack is popped and added to the list of parsed statements. flush-causing

`return(177)`	An empty statement is created and added to the list of parsed statements. flush-causing
`getstatic(178)`	FA(V(CPV), CPV, CPV) Accesses a field of a static class. Both the classname and the fieldname as also the field type are retrieved from the constant pool. Helper function: `__readfield`
`putstatic(179)`	A(FA(V(CPV), CPV, CPV), OSA1) Accesses a field of a static class. Both the classname and the fieldname as also the field type are retrieved from the constant pool. Helper function: `__writefield`
`getfield(180)`	FA(OSA1, CPV, CPV) The fieldname as also the field type are retrieved from the constant pool. Helper function: `__readfield`
`putfield(181)`	A(FA(OSA2, CPV, CPV), OSA1) The fieldname as also the field type are retrieved from the constant pool. Helper function: `__writefield`
`invokevirtual (182), invokespecial (183)`	A(V(tempN), MC(FA(OSAN, CPV, CPV), <OSA1, OSA2, ..., OSAN-1>, CPV)) Adds an Assignment of the result of MethodCall to a newly created Variable to the list of parsed statements. The temporarily created variable holding the result of the MethodCall is then pushed to the operand stack. The introduction of a variable is used to cope with possible side effects of methods. Helper function: `__invokemethod`

`invokestatic (184)`	A(V(tempN), MC(FA(V(CPV), CPV, CPV), <OSA1, OSA2, ..., OSAN-1>, CPV)) Adds an Assignment of the result of MethodCall to a newly created Variable to the list of parsed statements. The temporarily created variable holding the result of the MethodCall is then pushed to the operand stack. The introduction of a variable is used to cope with possible side effects of methods. Helper function: `__invokemethod`
`invokeinterface (185)`	A(V(tempN), MC(FA(OSAN, CPV, CPV), <OSA1, OSA2, ..., OSAN-1>, CPV)) Adds an Assignment of the result of MethodCall to a newly created Variable to the list of parsed statements. The temporarily created variable holding the result of the MethodCall is then pushed to the operand stack. The introduction of a variable is used to cope with possible side effects of methods. Helper function: `__invokemethod`
`new(187)`	N(CPV)
`newarray(188)`	MC(FA(N(BAV), 'initialize', BAV), <OSA1>, BAV)
`anewarray(189)`	MC(FA(N(CPT), 'initialize', CPV), <OSA1>, CPV)
`arraylength (190)`	MC(FA(OSA1, 'length', ()I), null, I)
`athrow(191)`	T(OSA1) flush-causing

`checkcast(192)`	MC(FA(OSA2, 'isInstanceOf', (Ljava/lang/String;)Z), <NC(CPV, Ljava/lang/String;)>, Z)
`instanceof(193)`	MC(FA(OSA2, 'isInstanceOf', (Ljava/lang/String;)Z), <NC(CPV, Ljava/lang/String;)>, Z)
`monitorenter (194)`	MC(FA(OSA1, 'monitorenter', ()V), null, V) flush-causing
`monitorexit (195)`	MC(FA(OSA1, 'monitorexit', ()V), null, V) flush-causing
`wide(196)`	No statement is created. The wide flag is set to true.
`multianewarray (197)`	MC(FA(N(CPT), 'initialize', CPV), <OSA1, OSA2, .., OSAN>, CPV)
`ifnull(198)`	MC(FA(OSA1, '==', (Ljava/lang/Object;)Z), <NC('null', Ljava/lang/Object;)>, Z) flush-causing
`ifnonnull(199)`	MC(FA(OSA1, '!=', '(Ljava/lang/Object;)Z), <NC('null', Ljava/lang/Object;)>, Z) flush-causing
`goto_w(200)`	Same as `goto` just more bytes used to determine address.
`jsr_w(201)`	Same as `jsr` just more bytes used to determine address.

In the following table an overview of the used and introduced method calls and their semantics is given.

MethodCall	**Semantics**
`x.+(y)`	`x + y`
`x.-(y)`	`x - y`
`x.*(y)`	`x * y`
`x./(y)`	`x / y`
`x.%(y)`	`x % y` Remainder of division
`x./(y)`	`x / y`
`x.negate()`	`0 - x`
`x.<<(y)`	`x << y` Arithmetic shift
`x.>>(y)`	`x >> y` Arithmetic shift
`x.>>>(y)`	`x >>> y` Logical shift
`x.&(y)`	`x & y` Bitwise and
`x.|(y)`	`x | y` Bitwise or
`x.^(y)`	`x ^ y` Xor
`x.apply(y)`	`x[y]`
`x.update(y,z)`	`x[y] = z`
`i.i2l()`	`(long)i` Type conversion from int to long
`i.i2f()`	`(float)i` Type conversion from int to float
`i.i2d()`	`(double)i` Type conversion from int to double
`l.l2i()`	`(int)l` Type conversion from long to int
`l.l2f()`	`(float)l` Type conversion from long to float
`l.l2d()`	`(double)l` Type conversion from long to double
`f.f2i()`	`(int)f` Type conversion from float to int
`f.f2l()`	`(long)f` Type conversion from float to long

`f.f2d()`	`(double)f` Type conversion from float to double
`d.d2i()`	`(int)d` Type conversion from double to int
`d.d2l()`	`(long)d` Type conversion from double to long
`d.d2f()`	`(float)d` Type conversion from double to float
`i.i2b()`	`(byte)i` Type conversion from int to byte
`i.i2c()`	`(char)i` Type conversion from int to char
`i.i2s()`	`(short)i` Type conversion from int to short
`x.compareg(y)`	If x greater than y the result is 1. If both values are equal the result is 0. If x is less than y the result is -1. If one of the values is NaN the result is 1.
`x.comparel(y)`	If x greater than y the result is 1. If both values are equal the result is 0. If x is less than y the result is -1. If one of the values is NaN the result is -1.
`x.compare(y)`	If x greater than y the result is 1. If both values are equal the result is 0. If x is less than y the result is -1.
`x.==(y)`	`x == y`
`x.!=(y)`	`x != y`
`x.<(y)`	`x < y`
`x.>=(y)`	`x >= y`
`x.>(y)`	`x > y`
`x.<=(y)`	`x <= y`
`x.initialize(q,k,...x)`	`x = new int[q][k]..[x]` Array initialization. Note that the type is not fixed to int.
`x.length()`	The length of the array is determined.
`x.isIstanceOf(y)`	`x instanceof y`
`x.monitorenter()`	Enter monitor for object.
`x.monitorexit()`	Exit monitor for object.

4. Conclusions

JavaToSimple was completely written in Java whereas Sample and also Simple were written in Scala. Nevertheless the interoperation of the two languages didn't cause too much trouble, as we expected it because Scala is built on top of Java. There are still some open issues. First of all the implementation of the JavaType class is very incomplete. One needs to implement the method it derives from the parent Type class. Secondly, until now no effort has been done trying to optimize for speed of the parsing proccess. That would be a possible point for improvements in the future, although the parsing process is not really slow. It is actually comparable in time with compilation. In the end all the Java opcodes described in the Java Virtual Machine Specification [2] were successfully implemented. To test the implementation we let JavaToSimple parse the whole Java Class Library containing over 16'000 classes which were all parsed without generating any errors. JavaToSimple is therefore able to parse most if not all correct Java classfiles and generate ClassDefinition's for them.

Bibliography

[1] Pietro Ferrara,
How to extend Sample to analyze a new language.
ETH Zurich 2010

[2] Tim Lindholm, Frank Yellin,
The Java Virtual Machine Specification, Second Edition
`http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`

[3] Shigeru Chiba,
Javassist
`http://www.csg.is.titech.ac.jp/ chiba/javassist/`