Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 141 (2005) 203–220
www.elsevier.com/locate/entcs

JVM Bytecode Verification Without Dataflow Analysis

Ian Bayley[1]

Department of Computing
Oxford Brookes University
Oxford OX33 1HX, U.K.

Sam Shiel[2]

Abstract

Bytecode verification algorithms are traditionally based on dataflow analysis. We present an alternative algorithm that first restructures the bytecode and then infers a type signature for each method in a manner typical of functional programming languages. We also give an operational semantics to an algebra of structured bytecode and thereby prove both that restructuring preserves semantics and that our type inference is sound.

Keywords: program transformation, bytecode verification, Java Virtual Machine

1 Introduction

The Java Virtual Machine (JVM) [6] consists of a register array (holding the current method's parameters and local variables), an operand stack (used for evaluating expressions) and a program counter (which points at the next instruction). The JVM changes this state by executing simple low-level instructions, which typically perform some useful operation on the top elements of the stack or move data between the stack and the registers.

[1] Email: ibayley@brookes.ac.uk
[2] Email: sam.shiel@orange.net

doi:10.1016/j.entcs.2005.02.029

A bytecode verifier interprets these instructions on an abstraction of the state that consists of an array of types and a stack of types, looking for instances where, to give just two examples, an instruction does not receive enough arguments of the right type from the operand stack or tries to read from a register in which nothing has yet been stored. This search is done by attempting to calculate entry types and exit types for each instruction, terminating with failure when this cannot be done consistently.

The main way to change the control flow is to execute a jump instruction, which conditionally or unconditionally (depending on the type of instruction) changes the value stored in the program counter. These jumps complicate verification significantly because each instruction can have several predecessors or several successors or even both. The entry type of each instruction can therefore influence and be influenced by that of many other instructions, so type checking cannot in general be done in a single pass. Instead, a worklist algorithm, in effect, traverses the control flow graph, calculating the entry type of each instruction as the most specific supertype of the exit types of its predecessors. See [5] for an accessible introduction.

Our alternative is to transform standard "unstructured" bytecode containing jump instructions into a "structured" (as in block-structured) bytecode without explicit jumps. Structured bytecode is composed of blocks, each of which has at most one predecessor and at most one successor. This makes it possible to dispense with the traditional data-flow analysis approach and to verify blocks by determining a type signature in a compositional manner from the block's sub-components. This approach has many advantages: rules can be stated without reference to the program counter, soundness of verification can be proven by a simple structural induction, and type checking is strengthened to inference. Most importantly, verification is given a new formal foundation on which to base further analysis.

Section 2 motivates, introduces and gives a semantics for an algebra of structured bytecode. Section 3 presents type inference rules for structured bytecode and proves a soundness theorem. Section 4 gives an algorithm for structuring unstructured bytecode and shows that the transformation is meaning-preserving. Unlike many structuring algorithms in the literature, this one is specialised to bytecode and for added simplicity can be applied without building first either a parse tree or a control flow graph. Finally, Section 5 discusses directions for future research and Section 6 concludes.

2 Structured bytecode

2.1 State Transitions

We first define some useful sets:

- $Word$, the set of 32-bit words. We shall not examine longs and doubles, which both take up two words, and we shall discuss types later.
- the set of operand stacks, given by $OperandStack = Word^*$
- the set of register arrays, given by $RegisterArray = Word^*$
- the set of program counter (pc) values, given by $Location = \mathbb{N}$

Here, X^* is the set of sequences formed from elements of X; so let x: xs denote the sequence formed by adding x to the start of xs, let $xs +\!\!+ ys$ denote the result of concatenating sequence xs to the front of sequence ys, and let $\#xs$ denote the number of elements in xs.

The state of the JVM is an element of *UState* where

$$UState = OperandStack \times RegisterArray \times Location$$

In syntactic terms, a method is a sequence of instructions and an instruction is an opcode followed by a fixed number of operands.

$$Method ::= \{Instruction\}$$

$$Instruction ::= Opcode, \{Operand\}$$

Semantically, methods and instructions are both transition relations on *UState*. More precisely, each instruction is a partial function, since it is deterministic, and we shall make it total by adding a stuck state $\otimes$ to *UState*. We define each transition relation with rules in the form of (*premise*, *conclusion*) pairs, where each *premise* is a predicate on the state and operands, and each *conclusion* is a transition performed when the premise holds. When no premise holds, the state becomes $\otimes$; a major objective of verification is to ensure that this cannot happen. Premises may contain type judgements and may also be given implicitly by pattern matching, as with the instruction `swap`, below

$$\frac{}{([top, next] +\!\!+ rest, regs, pc) \multimap [\texttt{swap}] \rightarrowtail ([next, top] +\!\!+ rest, regs, pc + 1)}$$

Here, the premise is that there must be at least two elements on the operand stack. We shall distinguish between *jumping* instructions and *non-jumping* instructions. The latter class are those instructions, like `swap`, that unconditionally add their length to the program counter. We shall, however, refrain from giving semantics to any more of these instructions, since we need to focus on the jumping instructions that do change the control flow.

2.2 Removing gotos in a high-level language

For a dialect of Java with `goto`s, the code fragment

```
label: x++; x=x*2;
if (x==0) goto label;
x=x*3; return;
```

could be rewritten without `goto`s as follows

```
do {x++;
    do {x*=2;
        do {if (x==0)
                goBackFlags = {true, false, false};
            else
            do {x*=3;
                goBackFlags = {false, false, false, false};
            } while (goBackFlags[3]);
        } while (goBackFlags[2]);
    } while (goBackFlags[1]);
} while (goBackFlags[0]);
```

Observe that instead of jumping directly to the statement `x++` when `(x==0)` is true, we fall through the inner loops and repeat the outer loop. Instead of the statement `return` (effectively a jump to the end) we fall through every one of the enclosing loops. The two different arrays assigned to `goBackFlags`, which dictate this behaviour, are built up by pushing onto an initially empty stack, a boolean for each while loop introduced; this boolean has the value `true` if the current instruction, and start of the loop, is the target of the jump and it is `false` otherwise. There is an obvious optimisation to produce less unwieldy code: insert a `do...while` loop only where the instruction considered is the target of a future jump. This would give

```
do {x++; x*=2;
    if (x==0) goBackFlags = {true};
    else {x*=3; goBackFlags = {false};}
} while (goBackFlags[0]);
```

The rewriting algorithm demonstrated here is inspired by the flowchart transformation given by Böhm and Jacopini in [3]. We have chosen it (above alternatives like [1] and [8]) for the sake of simplicity to make it relatively easy to show that our structuring preserves semantics. Notice that the array is assigned only once during each execution and that it always consists of an optional `true` followed by any number of `false`s. We shall formalise the algorithm in Section 4.

2.3 Algebraic definition of structured bytecode

The structured Java above suggests that structured bytecode should manipulate a state in $SState$ where (for $BoolStack = \mathbb{B}^*$)

$$SState = OperandStack \times RegisterArray \times BoolStack$$

The third component plays a role similar to the array `goBackFlags`. To write a member of $BoolStack$, we abbreviate `true` and `false` to T and F, sequence concatenation to juxtaposition, and use F^n to denote n consecutive Fs. For example, F^2T means "fall through the inner two loops and repeat the outermost", since the order is topmost first.

We use the word *block* to refer to any member of the language of structured bytecode L, which has cases for *boolean stack assignment*, *non-jumping instructions*, *sequence*, *selection*, and *iteration* as defined, in order, below. More formally, L is the smallest language such that

- if $bs \in BoolStack$ then $|bs| \in L$, with the interpretation "set the boolean stack to bs",
- for any non-jumping instruction in $UState \times UState$, form a corresponding instruction in $SState \times SState$ with the same syntax and an identical effect on the operand stack and register array but which leaves the boolean stack unchanged. The instruction formed is in L.
- if l_1 and l_2 belong to L then so does $l_1; l_2$, with the interpretation "do l_1 first then do l_2".
- if l_1 and l_2 belong to L then so does $\langle l_1, l_2 \rangle$, with the interpretation "if the top of the operand stack is zero then do l_1 else do l_2".
- if l belongs to L then so does $\{l\}$, with the interpretation "do l at least once and until the top of the boolean stack is F"

The last two cases both pop the stack they test. Note that no instruction can be sequenced after a boolean stack assignment.

2.4 Semantics of structured bytecode

We now define by structural induction, a transition relation for each term of the language. The first rule embeds non-jumping instructions into structured bytecode; let $len\ (i)$ denote the length in bytes of instruction i.

$$\frac{(os, regs, pc) \multimap [i] \rightarrowtail (os', regs', pc + len\ (i))}{(os, regs, bs) \stackrel{i}{\Longrightarrow} (os', regs', bs)}$$

$$\frac{}{(os, regs, [\,]) \stackrel{|bs|}{\Longrightarrow} (os, regs, bs)}$$

$$\frac{(os, regs, bs) \stackrel{l_1}{\Longrightarrow} (os', regs', bs') \quad (os', regs', bs') \stackrel{l_2}{\Longrightarrow} (os'', regs'', bs'')}{(os, regs, bs) \stackrel{l_1;l_2}{\Longrightarrow} (os'', regs'', bs'')}$$

$$\frac{o = 0 \quad (os, regs, bs) \stackrel{l_1}{\Longrightarrow} (os', regs', bs')}{(o : os, regs, bs) \stackrel{\langle l_1, l_2 \rangle}{\Longrightarrow} (os', regs', bs')}$$

$$\frac{o \neq 0 \quad (os, regs, bs) \stackrel{l_2}{\Longrightarrow} (os', regs', bs')}{(o : os, regs, bs) \stackrel{\langle l_1, l_2 \rangle}{\Longrightarrow} (os', regs', bs')}$$

$$\frac{(os, regs, bs) \stackrel{l}{\Longrightarrow} (os', regs', F : bs)}{(os, regs, bs) \stackrel{\{l\}}{\Longrightarrow} (os', regs', bs')}$$

$$\frac{(os, regs, bs) \stackrel{l}{\Longrightarrow} (os', regs', T : bs) \quad (os', regs', bs') \stackrel{\{l\}}{\Longrightarrow} (os'', regs'', bs'')}{(os, regs, bs) \stackrel{\{l\}}{\Longrightarrow} (os'', regs'', bs'')}$$

2.5 Practicalities

There are at least four ways to obtain structured bytecode, and we list them all as it is not immediately obvious which solution is best.

(i) In this paper, we shall apply the structuring algorithm of Section 4, which has the advantage of not making any assumptions about the compiler that produced the bytecode. This is unlike the other three methods, all of which have the added disadvantage that they add slightly to the volume of bytecode.

(ii) We could serialise structured bytecode within unstructured bytecode by using some of the unused opcodes starting at hex CA to delimit the different kinds of block. Further opcodes could push values onto the boolean stack. As this is a substantial change to the JVM instruction set, this approach is unlikely to be popular, though it will save us the cost of restructuring.

The idea behind the other two methods is to compile the original source code in such a way that the original block structure can easily be inferred from the bytecode.

(iii) We could define an extra attribute that stores the locations of every jump that translates a `do...while` loop, along with similar information for the other program constructs. The source and target of each such jump will delimit the block exactly. Defining custom attributes that do not change bytecode semantics is permitted by the JVM spec [6]. This method makes too many assumptions about the bytecode though, as we shall see, as does the next proposal.

(iv) We could invert the action of a known compiler (assuming that compilation is injective). For example, Sun's `javac` compiler translates each program construct with a characteristic pattern of forward and backward jumps that makes identification simple. All that is needed is a simple case analysis based on the direction of each jump and the nature (jumping or not) of the adjacent statements. It is not obvious that `break` and `continue` statements can always be identified, but we shall discuss a more serious objection later.

The last three methods require iteration cases that pop the operand stack instead of the boolean stack, which would now be superfluous. For example, we could define our iteration case as follows:

$$\frac{o' \neq 0 \quad (os, regs) \stackrel{l}{\Longrightarrow} (o' : os', regs')}{(os, regs) \stackrel{\{l\}}{\Longrightarrow} (os', regs')}$$

$$\frac{o' = 0 \quad (os, regs) \stackrel{l}{\Longrightarrow} (o' : os', regs') \quad (os', regs') \stackrel{\{l\}}{\Longrightarrow} (os'', regs'')}{(os, regs) \stackrel{\{l\}}{\Longrightarrow} (os'', regs'')}$$

Here, the loop test is for equality to zero, as the construct represents the loop formed by a backward branch with the `ifeq` opcode. An obvious generalisation would be to parameterise the iteration and selection constructs by other conditional branch opcodes. Further iteration constructs could be introduced for the other types of loop.

The serious problem with (iv), and to a lesser extent with (iii), is that attempting to rediscover the original structure of bytecode is one of the trickier subproblems of decompilation by another name. Unfortunately, as reported in [7], even the slighest change to a file such as a peephole optimisation can prevent successful decompilation so it is unlikely we could service a different compiler for Java, let alone another language. In conclusion, we have decided to adopt method (i).

3 Type Inference

3.1 Types

The set of JVM types, denoted *Type*, is a bounded lattice partially ordered by the subtype relation $\sqsubseteq$ given by assignment compatibility in Java. The top ($\top$) and bottom ($\bot$) elements represent uninitialised and unreached (ie not yet examined) slots respectively. For simplicity, we ignore both arrays and interfaces. The state, belonging to *SState*, is given a type in *StateType* where

$$StateType = Type^* \times Type^* \times \mathbb{N}$$

These components are, in order, the types on the operand stack, the types in the register array and the height of the boolean stack. Any unused slot in the register array has type $\perp$ and the length of the register array is given by the `max_locals` item in the `Code` attribute of the method. The ordering $\sqsubseteq$ is lifted pointwise to $Type^*$, and then to *StateType* by the following definition: (using & for logical conjunction)

$$(o, r, b) \sqsubseteq (o', r', b') \quad =_{def} \quad o \sqsubseteq o' \;\&\; r \sqsubseteq r' \;\&\; b \leq b'$$

The lub (least upper bound) and glb (greatest lower bound) operators $\vee$ and $\wedge$ are extended to *StateType* in the same way.

3.2 Type signatures

Since types are subsets of a universal set of values, we shall use the notation $x \in t$ to indicate that x belongs to type t. A block s is given a type signature of $T \to T'$ (written $s : T \to T'$) if

(i) when given an input $x \in T$ it will either loop or terminate in a state $y \in T'$ and

(ii) T is the most general type assignable to x: for any type U, if $x \in U$ then $U \sqsubseteq T$

(iii) T' is the least general type assignable to y: for any type U', if $x \in U'$ then $T' \sqsubseteq U'$

Conditions (ii) and (iii) are needed to make the notion of type signatures canonical since otherwise for any z, both $\perp \to z$ and $z \to \top$, for example, would be valid type signatures. Furthermore, note that both conditions are enforced by conventional verification algorithms based on data-flow analysis, since the source type of each instruction is the lub of the target type of its predecessors and the target type calculated from the source type is the most specific applicable.

3.3 Inference rules

The rules below are both defined and proven correct, with respect to the operation semantics, by induction on the structure of structured bytecode. The first base case, boolean stack assignment, is relatively straightforward.

$$\overline{|bs| : (o, r, 0) \to (o, r, \#bs)}$$

The other base case of non-jumping instructions is deferred, but for now it is sufficient to know that it satisfies all three properties.

Now for the inductive cases, starting with sequence. We use the abbreviation SIH to refer to the structural induction hypothesis. Suppose $fst : T_1 \to T_3$

is executed on a state $x \in T_1$, followed by $snd : T_3 \to T_2$. Then $fst; snd$ terminates only if both fst and snd do. If so, we can use SIH to conclude that $fst; snd$ terminates in a state $y \in T_2$. We can also use SIH to confirm that properties (ii) and (iii) hold.

$$\frac{fst : T_1 \to T_3 \quad snd : T_3 \to T_2}{fst; snd : T_1 \to T_2}$$

To help us state the rules for selection and iteration, we shall define two unary functions on $StateType$, O and B, which return the state produced by pushing a boolean onto the operand stack or boolean stack.

$$O\ (o, r, b) = ([\mathbb{B}] +\!\!+ o, r, b)$$
$$B\ (o, r, b) = (o, r, b + 1)$$

For the selection case, suppose $\langle then, else \rangle$ is executed on $x \in O\ (s \wedge u)$. If so, the block $then$ is executed on states in s and u. Suppose $then : s \to t$ and $then$ terminates. Then by SIH it does so with type t which we can relax to $t \vee v$. After applying similar reasoning to the block $else$, we obtain the rule

$$\frac{then : s \to t \quad else : u \to v}{\langle then, else \rangle : O\ (s \wedge u) \to t \vee v}$$

Note that it is properties (ii) and (iii) that lead us to choose the *greatest* lower bound and *least* upper bound.

Finally, the rule for the iteration case is

$$\frac{body : s \to B\ (s)}{\{body\} : s \to s}$$

To prove this, we use induction on the number of iterations, remembering that type signatures say nothing about iterations that do not terminate. The base case, of only one iteration, uses the rule that pops F and the inductive case uses the rule that pops T, together with SIH as one of the premises. Properties (ii) and (iii) are proven correct for the base case of the loop induction by SIH and for the inductive case by SIH again, along with the loop induction hypothesis and an argument similar to that for sequence above.

This law has been designed to reflect the behaviour of a conventional verifier. One helpful reading of it is that a loop cannot overflow (nor underflow) the stack if its body does not add to (nor take away from) the stack.

3.4 Inference rules with variables

Three problems are associated with giving type signatures to non-jumping instructions. First, we need to add guards on the source type to ensure, for example, that the operand stack does not overflow or underflow. This is mostly a notational matter, however, so we shall not explore it further. Secondly,

most instructions use part of but not all of the operand stack so most type signatures need a variable to represent the unused part of the operand stack.

For example, for `iadd` we effectively have a collection of type signatures, one for each value of $X \in \mathit{Type}^*$ (up to the allowed height), which is implicitly universally quantified.

$$\texttt{iadd} : ([\texttt{int},\texttt{int}] +\!\!+ X, r, b) \to ([\texttt{int}] +\!\!+ X, r, b)$$

Thirdly, many instructions can be applied to a type more specific than the source type of the type signature. So the law for sequence should be adapted as follows:

$$\frac{s_1 : T_1 +\!\!+ X \to T_1' +\!\!+ X \quad T_1' +\!\!+ X \sqsubseteq T_2 +\!\!+ Y \quad s_2 : T_2 +\!\!+ Y \to T_2' +\!\!+ Y}{s_1; s_2 : T_1 +\!\!+ X \to T_2' \, [\![T_2 \setminus T_1']\!] +\!\!+ Y}$$

Here, $T \, [\![T' \setminus T'']\!]$ denotes the state type T with every element type that occurs in T' replaced with the more specific type to which it is strengthened in T''.

If we write $s :: T \to T'$ to abbreviate any type signature of the form $s : T +\!\!+ X \to T' +\!\!+ X$ then we have (for some binary operator $*$ on type signatures)

$$\frac{s_1 :: T_1 \to T_1' \quad s_2 :: T_2 \to T_2'}{s_1; s_2 :: (T_1 \to T_1') * (T_2 \to T_2' \, [\![T_2 \setminus T_1']\!])}$$

To obtain a definition of $*$ we decompose the $\sqsubseteq$ inequality in the previous rule by case analysis on T_1' and T_2:

- if $T_1' = [\,]$ then $X \sqsubseteq T_2 +\!\!+ Y$ so we choose $X = T_2 +\!\!+ Y$ since the type signature must be canonical
- if $T_2 = [\,]$ then $T_1' +\!\!+ X \sqsubseteq Y$ so we choose $Y = T_1' +\!\!+ X$ since the type signature must be canonical
- if $T_1' = h_1' : t_1'$ and $T_2 = h_2 : t_2$ then $h_1' : t_1' +\!\!+ X \sqsubseteq h_2 : t_2 +\!\!+ Y$ so $h_1' \sqsubseteq h_2$ and $t_1' +\!\!+ X \sqsubseteq t_2 +\!\!+ Y$, since $\sqsubseteq$ is a lexicographic ordering.

This third case causes us to define $*$ recursively.

$$\begin{aligned}
(T_1 \to [\,]) * (T_2 \to T_2') &= T_1 +\!\!+ T_2 \to T_2' \\
(T_1 \to T_1') * ([\,] \to T_2') &= T_1 \to T_1' +\!\!+ T_2' \\
(T_1 \to h_1' : t_1') * (h_2 : t_2 \to T_2') &= (T_1 \to t_1') * (t_2 \to T_2') \text{ if } h_1' \sqsubseteq h_2
\end{aligned}$$

These laws were presented in a different and unmotivated form in [4] for an unusual bytecode-like language where instructions could only be combined by sequencing.

Finally, we shall generalise the rule for sequence to opcodes like `swap` that have variables in their type signatures. If T is a set of type variables, then let $\forall T \cdot X \to Y$ denote a type signature where every type variable in $X \to Y$ is

in T. If σ is a mapping that replaces variables with fresh variables and the variables of Z are a subset of σ's domain then $\sigma\ [Z]$ is the type produced by applying this mapping. Then for any σ and τ with disjoint ranges:

$$\frac{s_1 : T_1 +\!\!+ X \to T_1' +\!\!+ X \quad s_2 : T_2 +\!\!+ Y \to T_2' +\!\!+ Y \quad \sigma\ [T_1' +\!\!+ X] \sqsubseteq \tau\ [T_2 +\!\!+ Y]}{s_1; s_2 : \sigma\ [T_1 +\!\!+ X] \to \tau\ [T_2'\ [\![T_2 \setminus T_1']\!] +\!\!+ Y]}$$

This specialises to the law we already derived for sequence. The rules for selection and iteration do not need to be revisited since there is no potential clash of variables.

4 Structuring unstructured bytecode

4.1 Semantics of unstructured bytecode

Having defined the semantics of structured bytecode, we must now do the same for unstructured bytecode. An *unstructured method* is a mapping from locations to instructions, where the start location is 0. We use the notation u_T to refer to the instruction at location T in u, if it exists. Below, the identifier u will range over unstructured methods.

An *instruction* is a function on *USState* where

$$USState = OperandStack \times RegisterArray \times BoolStack \times Location$$

We use the word *target* to describe an assignment to either the boolean stack or the pc. Unstructured bytecode has only the latter sort of target but our structuring algorithm uses both for its intermediate form. To bridge the two states, let s_T be the member of *USState* formed by affixing a location T to $s \in SState$.

As well as the non-jumping instructions, which ignore the boolean stack and add their length to the pc, we have three (abbreviated) jumping instructions based on `ifeq`, `goto`, and `return`:

- I_f^t, which performs target t if the operand stack is zero and target f if it is not,
- G^t, which performs target t unconditionally, and
- R^t, which performs target t, strictly a boolean stack, and changes the pc to a notional value *end* representing termination.

Using identifiers pc,pc' for locations and bs,bs' for boolean stacks, the operational semantics for these instructions are as follows:

$$\frac{}{(os, regs, pc, bs) \dashv[G^{pc'}]\!\!\rightarrowtail (os, regs, pc', bs)}$$

$$\frac{}{(os, regs, pc, bs) \dashv[G^{bs'}]\!\!\rightarrowtail (os, regs, pc, bs')}$$

$$\frac{o = 0}{(o : os, regs, pc, bs) \dashv[I_f^{pc'}]\vdash (os, regs, pc', bs)}$$

$$\frac{o = 0}{(o : os, regs, pc, bs) \dashv[I_f^{bs'}]\vdash (os, regs, pc, bs')}$$

$$\frac{o \neq 0}{(o : os, regs, pc, bs) \dashv[I_{pc'}^{t}]\vdash (os, regs, pc', bs)}$$

$$\frac{o \neq 0}{(o : os, regs, pc, bs) \dashv[I_{bs'}^{t}]\vdash (os, regs, pc, bs')}$$

$$\frac{}{(os, regs, pc, bs) \dashv[R^{bs'}]\vdash (os, regs, end, bs')}$$

We also subscript the non-jumping instructions with targets, which initially hold the location of the lexically next instruction; we ensure the final instruction is an R. We write $\underline{f}$ to denote the function that applies f to every target in an unstructured method.

The effect of executing u starting at location T is given by $\chi_{T,u}$, a partial function on $SState$, which acts as the identity function when $T = end$ and everywhere else as follows:

$$\chi_{T,u}\ \ s = s' \quad =_{def} \quad \exists s'', T' \bullet s_T \dashv[u_T]\vdash s''_{T'}\ \&\ \chi_{T',u}\ \ s'' = s'\ \&\ T \neq end$$

4.2 The structuring algorithm

The result of structuring u is given by $\sigma_{0,u}$ where

$$\sigma_{P,u} = \begin{cases} \{i_T; \phi_T\ (\underline{\mu_P}\ u)\} & \text{if } u_P = i_T \text{ , non-jumping } i \\ \{\langle \phi_T\ (\underline{\mu_P}\ u), \phi_F\ (\underline{\mu_P}\ u)\rangle\} & \text{if } u_P = I_F^T, P \neq T, P \neq F \\ \{\phi_T\ (\underline{\mu_P}\ u)\} & \text{if } u_P = G^T, T \neq P \\ |bs| & \text{if } u_P = R^{bs} \end{cases}$$

Auxiliary functions μ_P and ϕ_T are defined as follows. Given a location target that matches P, μ_P creates a new target $[T]$ to repeat the loop just introduced; all other locations are left unchanged. Given a boolean stack target, μ_P pushes F to quit the same loop.

$$\mu_P\ bs = [F] +\!\!+ bs$$
$$\mu_P\ P = [T]$$

Now, ϕ_T is a function on unstructured bytecode that either terminates with a boolean stack assignment or moves to a new location, depending on the type of the target T.

$$\phi_{pc}\ u = \sigma_{pc}\ u$$

$\phi_{bs}\ u = |bs|$

The algorithm takes quadratic time though we shall see later that it can be made linear. Other points to note are:

(i) unstructured methods can be implemented as sequences if "fillers" are inserted after each instruction that is longer than a byte.

(ii) the function μ removes P as a target and this ensures that mutual recursion between ϕ and σ is well-founded. So do the guards for case I.

(iii) the guard for case G is a convenience only, so that we need not translate infinite loops.

(iv) we must check that each location is valid before we move to it and we should only introduce a loop when the current instruction is the predecessor of the remaining instructions. We have omitted both of these checks for simplicity.

Now for a demonstration, based on the running example introduced in Section 2. Suppose a, b and c are non-jumping instructions. If we write σ's second parameter without subscripting, for added visibility, and abbreviate sequence by juxtaposition, since it is associative, we have:

$$\begin{aligned}
&\sigma_0[a_1, b_2, I_3^0, c_4, R^{[\,]}] \\
=\ &\{\mathbf{a}\ \sigma_1[a_1, b_2, I_3^B, c_4, R^F]\} \\
=\ &\{\mathbf{a}\{\mathbf{b}\ \sigma_2[a_1, b_2, I_3^{FB}, c_4, R^{F^2}]\}\} \\
=\ &\{\mathbf{a}\{\mathbf{b}\{\langle|\mathbf{F^2B}|,\sigma_3[a_1, b_2, I_3^{F^2B}, c_4, R^{F^3}]\rangle\}\}\} \\
=\ &\{\mathbf{a}\{\mathbf{b}\{\langle|\mathbf{F^2B}|, \{\mathbf{c}\ \sigma_4[a_1, b_2, I_3^{F^3B}, c_4, R^{F^4}]\}\rangle\}\}\} \\
=\ &\{\mathbf{a}\{\mathbf{b}\{\langle|\mathbf{F^2B}|, \{\mathbf{c}|\mathbf{F^4}|\}\rangle\}\}\}
\end{aligned}$$

4.3 Structuring preserves semantics

We prove the following three properties.

(i) $\sigma_{T,u}\ s$ has an empty boolean stack if it halts

(ii) $\sigma_{T,u}\ s$ halts whenever $\chi_{T,u}\ s$ halts

(iii) whenever $\sigma_{T,u}\ s$ halts it is equal to $\chi_{T,u}\ s$

Properties **(i)** and **(ii)** are lemmas used in the proof of **(iii)**. Property **(i)** derives some initial constraints on u and s.

The proofs are as follows:

(i) we use induction on the number of transitions. For the base case where $u_T = R^{bs}$ we require $bs = [\,]$. For the inductive case, we require that every target in u is a location or is $[\,]$, and we observe that the number of enclosing

loops matches the length of the eventual boolean stack assignment because each target is lengthened by one for each loop introduced.

(ii) if $\chi_{T,u}\ s$ loops then $\sigma_{T,u}\ s$ loops too because any jump is translated to a loop whenever T is pushed. If $\sigma_{T,u}\ s$ loops then a T must have been pushed but this can only happen when there is a jump to the current instruction forming a loop and so $\chi_{T,u}\ s$ loops too.

(iii) we use induction on the number of transitions. If $\sigma_{T,u}\ s$ halts then by property **(ii)** we know that $\chi_{T,u}\ s$ halts also. For the base case, where $u_T = R^{bs}$, both $\chi_{T,u}\ s$ and $\sigma_{T,u}\ s$ set the boolean stack to bs and the pc to end.

Now suppose u_T is a non-jumping instruction i and the block produced is $\{i_T; \phi_T\ (\underline{\mu_P}\ u)\}$. We just need to prove that a single iteration of the body does not change the final state in which the block finishes. The induction hypothesis will then make this true for several iterations. Suppose that i_T changes the state to s'. If this is so, then by the definition of ϕ, we have that $\phi_T\ (\underline{\mu_P}\ u)\ s'$ can either be a boolean stack assignment or a call to $\sigma_{T,u}\ (\underline{\mu_P}\ s')$.

The latter is equal to $\chi_{T,u}\ (\underline{\mu_P}\ s')$ by the induction hypothesis. We want to show that following this with a pop of the boolean stack and a jump to the relevant location gives a state equal to $\chi_{T,u}\ s'$. At this point we use the definition of χ to consider the two possible values that have been pushed onto each boolean stack target by μ_P. Whether T or F has been pushed, the pc is changed appropriately, and the loop is either repeated or exited.

We must still consider the case where $\phi_T\ (\underline{\mu_P}\ u)\ s'$ is a boolean stack assignment. By induction on the number of instructions visited so far by the structuring algorithm, we can be sure that the targets have been updated correctly at every stage before the boolean stack assignment was performed.

5 Further Work

5.1 Improvements to the structuring algorithm

The algorithm presented above was chosen to make the proof as concise as possible, subject to the constraint that we wished to reason at the low level of bytecode instructions rather than with flowcharts as Boehm and Jacopini had done. Having implemented the algorithm in Haskell, we are attempting to construct a more concise proof that derives the algorithm by equational reasoning from the semantics of both bytecodes. It is noteworthy that boolean stack assignments are used only to quit the method and to jump to the start of an enclosing iteration construct, so specialised instructions similar to Java's

`continue` and `return` can be used for clarity. Instead of assigning the boolean stack, we would write to a status variable particular values that convey information such as "returning" and "continuing to nth nested loop".

The algorithm takes quadratic time (in the number of instructions) because for each instruction considered i, every other instruction j is examined to find out if j is a predecessor of i. If it is then a loop is introduced at point i and the other boolean stack assignment targets are updated. We can make the algorithm linear by constructing a predecessors list for each instruction and comparing that with a list of unvisited locations when deciding whether to introduce a loop. When that happens, the location should be added to a list of loop starts, which is consulted later when the jump is encountered. If the jump is in the list, it is translated as one of the `continue` instructions. If the jump is not in the list, then it is followed. Each list can be consulted and updated in constant time, if it is stored as an array, at the cost of linear space. If this is deemed unacceptable, an optimisation could be to store in the predecessors list only the predecessors for instructions reachable through jumps.

5.2 Categorical semantics for structured bytecode

The iteration case can be implemented in Haskell as follows (for some data type `SState`):

```
doWhileTopBoolTrue :: (SState -> SState) -> (SState -> SState)
doWhileTopBoolTrue body = fold id id . unfold topBoolTrue id body . body
    where topBoolTrue (os, regs, bool:bools) = bool

data LoopResult a = TestFalse a | TestTrue (LoopResult a)

unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> (a -> LoopResult b)
unfold p f g x = if p x then TestFalse (f x)
                        else TestTrue (unfold p f g (g x))

fold :: (a -> b) -> (b -> b) -> LoopResult a -> b
fold f t (TestFalse x) = f x
fold f t (TestTrue x) = t (fold f t x)
```

This unusual definition exploits the strong connection between functional programming and category theory that is explored in [2], where theorems are defined for common forms of repetition called folds and unfolds. These theorems are direct consequences of the semantics of algebraic datatypes. For example, fusion laws can be used to move code in and out of loops. The base category used would have members of `SState` as objects, and partial functions on `SState` as arrows. Composition of arrows is given by reverse-order sequencing with ';', which is associative with the equivalent of `nop` for *SState* as a unit. So this is indeed a category as required. Furthermore, the selection construct also has a categorical foundation in the notion of a

coproduct. Alternatively, we can attempt to derive such simple algebraic laws from the operational semantics directly.

5.3 Major challenges of verification

We now consider method type signatures, and the major issues of object initialisation, subroutines and exceptions. Solutions are only sketched, due to lack of space and will be developed at a later date.

A type signature for the method as a whole can be inferred by looking at the initial type of the register array and method return instructions. This should then be matched against the given type descriptor, ignoring register 0 if the method is non-static. A method may ignore a parameter but it must not use an undeclared parameter. Since a type descriptor offers guarantees to a calling method, parameters must not be more general than inferred, nor may return types be less general. Type inference rules can easily be defined for the method invocation opcodes and typed return opcodes like `areturn`. The rule for iteration must be relaxed though to ensure that if the status is "returning", only the top of the operand stack should be examined and only then if the return type is not void.

Object creation and initialisation are two separate processes in the JVM. An obvious way to ensure that no uninitialised object is used in a method call is to give it a distinguished type, but this approach does not work when the object is created in a `try` block, from which it may quit early, or in an iteration, where only some of the created objects may be initialised. Extra conditions on object creation are imposed by the JVM spec [6] and we shall attempt to derive them by stating simple algebraic laws for iteration and `try` blocks.

Each subroutine [9], i.e. each target of a `jsr` (jump subroutine) instruction, can be given several type signatures, one for each register used by a `ret` (return from subroutine) instruction. (Multiple uses of the same `ret` instruction would be united into one exit by the structuring algorithm.) This could then be used as the type signature for the original `jsr` instruction with the extra condition that the return address used is the address of the next instruction. The notion of addresses, invaluable when we had a pc, will be retained for this purpose alone. The type system itself would enforce the crucial last-in-first-out convention, because we can only return to return addresses that are currently stored in local registers. It is also worth noting that used and unused registers are automatically distinguished since the latter will have no inferred type. It is possible that this approach will permit recursive subroutines, which will be useful in the compilation of functional programming languages.

Protected regions (the bytecode equivalent of `try` blocks), which are iden-

tified and delimited by an exceptions table in the bytecode itself, can be given two type signatures: one for normal exits and one for abnormal exits, where the whole operand stack is wiped but for the exception object. Multiple non-exceptional exits are united by structuring but the exceptional exits, which can come from any instruction, are not, unless they have the same handler, in which case their local variable types would need to agree. The storing of uninitialised objects in registers is banned as in the JVM specification itself.

Other features of verification that appear orthogonal to our approach but still worth considering for completeness include monitors, dynamic class loading, arrays, interfaces and the `invokestatic` instruction.

6 Conclusion

We have demonstrated that if bytecode is given a block structure then verification can be executed as a simple structural induction and proven to be sound by the same technique. The small subset of JVM bytecode that we have chosen to demonstrate this idea can, in principle, be extended to the full language, and we intend to do so. It can also be extended to other virtual machines including the Common Language Runtime. We also intend to extend, simplify and improve the efficiency of the structuring algorithm, which is needed to obtain structured bytecode in the first place.

7 Acknowledgements

We would like to thank the anonymous referees, for pointing out some shortcomings in the original draft of this paper, and Peter Knaggs, for pointing us to his paper on stack effects.

References

[1] E. Ashcroft and Z. Manna. Translating program schemas to while-schemas. *SIAM Journal of Computing*, 4(2):125–146, 1975.

[2] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[3] C. Boehm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, pages 366–371, 1966.

[4] Peter Knaggs and William Stoddart. Type inference in stack based languages. *Formal Aspects of Computing*, 5(4):289–98, 1993.

[5] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1997.

[7] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *CC*, pages 111–127, 2002.

[8] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[9] Raymie Stata and Martin Abadi. A type system for java bytecode subroutines. *ACM Trans. Program. Lang. Syst.*, 21(1):90–137, 1999.