

# Proof for Optimization: Programming Logic Support for Java JIT Compilers

Claire L. Quigley

Department of Computing Science, University of Glasgow,  
17 Lilybank Gardens, Glasgow, G12 8RZ, Scotland.  
claire@dcs.gla.ac.uk

**Abstract:** *One of Java's weakest areas is its low execution speed in comparison to compiled languages like C. The mobile nature of bytecode adds to the problem, as many checks are necessary to ensure that downloaded code from untrusted sources is rendered as safe as possible. The use of the Bytecode Verifier and JIT compilers can improve the performance of Java code, but both offer limited possibilities for optimization. We propose one approach to this problem, namely proving properties of bytecode programs that allow extra optimizations to be made, or facilitate existing optimizations. This paper describes the progress made so far in developing a Bytecode Programming Logic for this purpose.*

## 1 Introduction

One significant disadvantage of interpreted bytecode languages, such as Java, is their low execution speed in comparison to compiled languages like C. The mobile nature of bytecode adds to the problem, as many checks are necessary to ensure that downloaded code from untrusted sources is rendered as safe as possible. Despite these drawbacks, there do exist ways of reducing the negative aspects of such systems.

One approach is to carry out static type checking at load time, as in the case of the Java Bytecode Verifier. This reduces the number of runtime checks that must be done and also allows certain instructions to be replaced by `quick` instructions, which are implemented more efficiently. Another approach is the use of a Just In Time (JIT) Compiler that takes the bytecode and produces corresponding native code at runtime.

The JIT may produce a straightforward translation of bytecode to native code, or it may apply a range of optimizing transformations, from simply inlining the code (removing the overhead of the interpreter implementation) in basic JITs to more complex techniques like copy propagation, assertion merging, live variable analysis, dead code elimination, strength reduction, common subexpression elimination, and loop unrolling [5, 6].

There are, however, limits to the amount of optimization that can safely be done by the Verifier and JITs; some operations simply cannot be carried out safely without a certain amount of runtime checking. But what if it were possible to prove that the conditions the runtime checks guard against would never arise in a particular piece of code? In

this case it might well be possible to dispense with these checks altogether, allowing optimizations not feasible at present. In addition to this, because of time constraints, current JIT compilers tend to produce acceptable code as quickly as possible, rather than producing the best code possible. By removing the burden of analysis from them it may be possible to change this.

The Annotated JIT project [11], which is described in more detail in Section 2.2, attempts to combat these difficulties by developing an **annotating compiler**. In addition to producing bytecode from Java source code, this compiler annotates the resulting class file with information obtained during analysis of the code. The annotations can be used by an annotation-aware JIT compiler (AJIT) to produce faster native code—and to produce it more quickly—than would be possible for a standard JIT compiler.

But this raises the question. If the AJIT is capable of increasing the performance of JIT compilers so effectively, why is it necessary to approach the problem from a formal point of view? The answer lies in the concept of **trust**. A user receiving annotated class files must simply trust that the producer has not annotated the file in such a way that the AJIT will generate unsafe code. Such a situation may occur as the result of errors in the optimising compiler leading to incorrect annotations, or as a result of a deliberate attack on the integrity of the user's system.

A formal proof of the correctness the annotations applying to a particular program obviously eliminates the first problem and may also help to avoid the second perhaps by utilising a method along the lines of Proof Carrying Code [13]. Users could be reassured that the annotations attached to a class file have been formally verified to preserve certain properties—note that this does not mean that the program has been proved to be **correct**, only that it does or does not do certain things. This 'lightweight' approach to program verification is the basis of the Java Extended Static Checking project [8]. In addition to this, users could be supplied with some representation of the proof which they could themselves check, or perhaps (with time constraints in mind) the program could be digitally signed.

Keeping in mind the aim of making Java programs run faster, we are developing a programming logic for bytecode programs which will allow properties of bytecode programs pertaining to the optimisations mentioned above to be formally verified. Based on an inductive definition of an execution relation for sequences of bytecode, it will include rules for common patterns of bytecode instructions.

Section 2 of this paper gives a brief summary of the work on which our programming logic is based; Sections 3–6 describe the development of the logic in some detail and outlines the current state of the work; and Section 7 is concerned with other possible applications for the logic.

## 2 Related Work

Two pieces of work have provided a basis for our own research: Cornelia Pusch's formalisation of the JVM in Isabelle/HOL [15], and the Annotated JIT project [11].

## 2.1 An Operational Semantics for Java

In her paper *Formalizing the Java Virtual Machine in Isabelle/HOL* [15], Cornelia Pusch describes her formalisation of the JVM in the theorem prover Isabelle (using the HOL object logic) [14]. The author's aim is to provide a formal version of the Java Virtual Machine (JVM) Specification [12] which is not prey to the ambiguities and inconsistencies which tend to creep into informal specifications (and indeed do in the case of the JVM Specification). As this is a not inconsiderable undertaking, the theorem prover Isabelle is used to ensure a degree of reliability not likely to be achieved in a proof 'by hand'. Although a large subset of the Java language is formalised, there are areas which are not treated in this preliminary implementation; these include exception handling and dynamic class loading.

The paper outlines the formalisation of both static aspects of Java programs, such as well-formedness of class files, and relations between classes; and properties of the Java run-time system including object initialisation and the JVM heap. The author also describes an operational semantics for the subset of the JVM instruction set considered so far. This work is part of the *Bali* project [3] which is concerned with formalization of various aspects of the Java language in Isabelle/HOL. It makes an important contribution to the new VerifiCard project [4] coordinated by the LOOP Project [2] which is also involved in the formalization of Java. VerifiCard aims to formalize aspects of JavaCard [1], a subset of the Java language used on smartcards, and will include proofs of properties at the bytecode level.

My own work uses Pusch's formalisation of the JVM as a platform on which to build a programming logic for bytecode programs. This, along with certain extensions to the instruction set covered by Pusch, is described in detail in Section 3 of this report.

## 2.2 Annotating the Java Bytecodes in Support of Optimization

In [11] the authors begin by observing that while Java provides a portable, platform-independent stack machine, it does so at the expense of execution speed, as stack machines do not map well onto today's CPUs, which rely heavily on the use of register and caches for speed. In addition to having no concept of registers, Java bytecodes are also unable to express optimizations like instruction scheduling, elimination of runtime checks, strength reduction, and automatic reclamation of memory.

With the goal of achieving C-like performance while retaining bytecode's portability and preserving compatibility with existing JVMs, the authors have developed an **annotating compiler**. This behaves initially like a traditional optimizing compiler, analysing the code and performing optimizations before emitting bytecode. But rather than discarding the information produced by the analysis, as would normally be the case, the compiler attaches the relevant information to each emitted bytecode in the form of an annotation.

The annotations contain information on register allocation, memory disambiguation, memory reclamation, and run-time checking that would normally have to be recomputed by the JVM—and in some instances might not be possible to recompute from

the bytecode as too much information may have been lost. Annotations are stored separately from the bytecode in a classfile in order not to interfere with the running of the program on standard JVMs. A JVM with an annotation aware JIT compiler, however, can use the annotations to produce code that is closer to optimal more quickly.

### 3 A Programming Logic for Bytecode

In order to prove properties of bytecode programs, it was necessary to develop a logical framework that would support this. It was decided to build a programming logic, similar to that developed by C.A.R. Hoare for reasoning about properties written in a simple imperative language [10], but for bytecode programs.

The Hoare logic is based on the relation

$$\{P\} C \{Q\} \equiv \forall \sigma \sigma'. ((P(\sigma) \wedge \sigma, C \rightarrow \sigma') \supset Q(\sigma'))$$

where  $P$  and  $Q$  are pre and post conditions respectively,  $\sigma$  and  $\sigma'$  are states and  $C$  is a command in the language. The equation above states that if a predicate  $P$  holds in state  $\sigma$  and the command  $C$  is executed in  $\sigma$ , and execution terminates, the resultant state will be  $\sigma'$  in which the predicate  $Q$  will hold. This relation, in which termination is not guaranteed, is known as *partial correctness*; a similar relation in which termination is guaranteed is called *total correctness*.

The decision to prove properties of the bytecode programs themselves, rather than the corresponding Java source was made based on two main factors:

- Java programs are downloaded by consumers as bytecode, not source
- It is perfectly feasible (albeit not common in practice) to produce Java bytecode from another high level language, e.g. C, ML.

But this introduced several new problems, the fact that bytecode is ‘flat’ and contains `goto` instructions presented difficulties not encountered in the standard logic which dealt with a structured programming language.

The Hoare logic has three main components, however, which it seemed could be applied to bytecode programs, namely:

- Definition of the evaluation of a section of code in the language (based on the operational semantics)
- Definition of a pre- and post-condition relation on execution of code.
- Higher level rules for combining patterns of code

The development of a bytecode programming logic based on these components is described in the rest of this chapter.

## 4 Extending the Semantics

As Pusch’s work formalised a subset of the JVM, certain instructions are omitted, including all arithmetic instructions, such as `iadd`, `isub`, etc. In order to prove properties of real programs, however trivial, it was necessary to extend the model to include this class of instructions.

Initially, the instructions: `iadd` add the two integers at the top of the stack, and `inc var val` increment local variable `var` by integer value `val` were added to the existing Load and Store instructions of the model.

Problems were also encountered with the representation of branching instructions. In Java bytecode these are absolute jumps to a label, and in Pusch’s model are represented by relative branches, where the new value of the program counter is obtained by adding an offset to the current value. This offset is positive for a branch forward, negative for a branch backwards.

While this convention appears to have been eminently suitable for Pusch’s higher-level proofs, we experienced several difficulties while using it to reason about lower-level properties. In particular, problems arose with proofs involving branches backwards where a negative integer was added to the program counter (a natural number cast to an integer) and the result was then cast to a natural. All this type-casting rendered the proofs in Isabelle very awkward and it was decided to abandon the representation of a backwards branch with a negative offset.

In place of the two varieties of branching instructions in Pusch’s model—`conditional_branch` and `unconditional_branch`—there are now four branching instructions: `conditional_branch_fwd`, `conditional_branch_bwd`, `unconditional_branch_fwd`, and `unconditional_branch_bwd`, all of which take a natural number as offset, which is either added or subtracted to the current program counter depending on whether the branch is forwards or backwards. This keeps all branching proofs in the realm of natural number arithmetic and simplifies matters considerably.

### 4.1 Execution of a Sequence of Bytecode

One point that needed to be clarified was the actual meaning of  $\{P\} xs \{Q\}$  in the context of a bytecode program. In the standard Hoare logic, execution begins at the start of the sequence of commands, `xs`, and finishes at the end (assuming the program terminates). But with bytecode there is the possibility of jumping **into** the code at some point after the start of `xs` and **out** at a point before the end of `xs`.

The first thing to consider was the very basic question of how to define execution of a sequence of bytecode instructions. A straightforward recursive definition of the form

$$\begin{aligned} exec^* [ ] \sigma &= \sigma \\ exec^* (x :: xs) \sigma &= exec^* xs (exec x \sigma) \end{aligned}$$

was not feasible, as the execution of  $xs$  would not necessarily be linear—execution might well jump back to the beginning of  $xs$  after a few instructions. Pusch’s formalisation recognises this by defining execution of several bytecode instructions as the reflexive transitive closure of a series of single execution steps

$$\begin{aligned} exec\_all &:: [bytecode, jvm\_state, jvm\_state] \longrightarrow bool \\ CFS \vdash s - jvm \rightarrow t &\equiv (s, t) : \{(s, t). exec(CFS, s) = Some\ t\}^* \end{aligned}$$

But this only tells us whether a pair of states is in the set of pairs of states defined by this relationship. To enable us to define a partial correctness relation we need to know that, for a sequence of bytecode instructions, if we start executing in state  $\sigma$  we will finish execution in state  $\sigma'$ . This brought us back to the very important question: how do we define ‘finishing’ execution of a sequence of bytecode instructions?

One way to do this is to state that execution of a sequence of instructions has finished **when the program counter is no longer pointing into the sequence**. This led to an inductive definition of a relation describing the execution of a list of bytecode instructions, identified within a classfile by a **start** position and a **finish** position. If execution began in state  $\sigma$ , it would result in state  $\sigma'$ , where the program counter of  $\sigma'$  was outside the section delimited by **start** and **finish**.

Once this definition of execution, symbolised by  $\longrightarrow$ , was in place, it was possible to base the Hoare-like relation  $\{P\} xs \{Q\}$  on it. This provided a more solid base from which rules such as that for loops could be proved. The two definitions are described in more detail below.

## 5 The `exec_block` ( $\longrightarrow$ ) Relation

- $CFS$  is a set of Classfiles
- $s$  and  $f$  are triples of the form  $(classname, method\ locator, program\ counter)$  each allowing identification of a single instruction in  $CFS$ .
- $\sigma$  and  $\sigma'$  are states, each consisting of  $(exception\ option, heap, frame\ stack\ list)$ .

The relation  $\langle CFS, \sigma \rangle \xrightarrow[s]{s} \sigma'$  is **True** if executing the sequence of instructions in  $CFS$  that begins at the instruction identified by  $s$  and ends at the instruction identified by  $f$ , in the state  $\sigma$ , results in the state  $\sigma'$ , where the instruction identified by  $f$  is not contained the sequence of instructions in  $CFS$  bounded by  $s$  and  $f$ . Also  $pc(s) \leq pc(f)$ ,  $pc(s) \leq pc(\sigma)$ , and  $s$  and  $f$  should identify instructions in the same method of the same class. The relation is described by the rules:

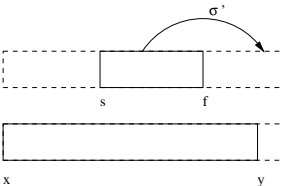
$$\begin{aligned} &exec(CFS, \sigma) = \sigma'; \\ &pc(s) \leq pc(\sigma) \leq pc(f); \\ &same\_method\ s\ \sigma\ \sigma'\ f; \\ &pc(f) < length(method\ CFS\ s); \\ &\frac{pc(\sigma') < pc(s) \vee pc(f) < pc(\sigma')}{\langle CFS, \sigma \rangle \xrightarrow[s]{s} \sigma'} \quad (1) \end{aligned}$$

$$\begin{array}{l}
 exec(CFS, \sigma) = \sigma''; \\
 pc(s) \leq pc(\sigma) \leq pc(f); \\
 same\_method\ s\ \sigma\ \sigma''\ f; \\
 pc(f) < length(method\ CFS\ s); \\
 \frac{\langle CFS, \sigma'' \rangle \xrightarrow[s]{s} \sigma'}{\langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma'} \quad (2)
 \end{array}$$

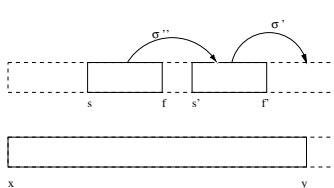
with (1) referring to the case in which one step of execution results in the program counter being outside the sequence of instructions under consideration; (2) the case where, after one step of execution, the program counter is still within the block of code delimited by  $s$  and  $f$ .

**Derived Theorems for  $\longrightarrow$**  Working with these rules, several properties were proved for the  $\longrightarrow$  relation.

**Theorem 1** Given a block in the relation and the position of the program counter on exit  $pc(\sigma')$ , it is possible to extend the relation to include all instructions in the method on the opposite side of the block from  $pc(\sigma')$ , and all instructions up to it on the same side.

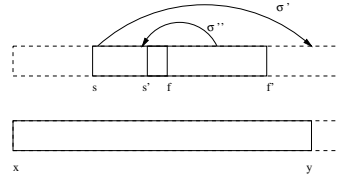
$$\begin{array}{l}
 \langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma'; \\
 y < pc(\sigma'); x < s; f < y; \\
 y < length(method\ CFS\ s) \\
 \hline
 \langle CFS, \sigma \rangle \xrightarrow[y]{x} \sigma' \quad (3)
 \end{array}$$


**Theorem 2** Add two blocks with gap in middle, extend on both sides.

$$\begin{array}{l}
 \langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma''; \\
 \langle CFS, \sigma'' \rangle \xrightarrow[f']{s'} \sigma'; \\
 f < s'; y < pc(\sigma'); \\
 x < s; f' < y; \\
 y < length(method\ CFS\ s) \\
 \hline
 \langle CFS, \sigma \rangle \xrightarrow[y]{x} \sigma' \quad (4)
 \end{array}$$


**Theorem 3** Add two blocks with overlap, extend on both sides.

$$\begin{array}{l}
 \langle CFS, \sigma'' \rangle \xrightarrow[f]{s} \sigma'; \\
 \langle CFS, \sigma \rangle \xrightarrow[f']{s'} \sigma''; \\
 s \leq s'; f \leq f'; \\
 y < pc(\sigma'); x < s; f' < y; \\
 y < \text{length}(\text{method } CFS \ s) \\
 \hline
 \langle CFS, \sigma \rangle \xrightarrow[y]{x} \sigma'
 \end{array} \quad (5)$$



**Theorem 4** A block of size one is equivalent to execution of that instruction, providing the instruction is not a “degenerate” branch (one which branches back to itself, or branches outside the bounds of the method), or an instruction which results in a frame being pushed or popped from the stack, i.e. method invocation or return. The later constraint is due to the current, simplified, definition of `exec_block` which demands that the initial and final states be in the same method.

The predicate on branching instructions is due to the fact that it would be possible to create a bytecode program (though probably not via a conventional compiler), which pushed the value `Null` onto the stack, followed by some other values, then had a branch if not null instruction which looped back to itself. Initially the top value on the stack would not be `Null`, but every time the branch instruction was evaluated, the top value on the stack would be popped, leading eventually to a state where the top value **was** `Null` and the loop was exited. This would mean that this instruction was in the `exec_block` relation, but that this instance of the relation was **not** equivalent to a single-step execution of the initial state.

$$\begin{array}{l}
 \text{get\_instr}(CFS, s) = x; \\
 \text{not\_degenerate\_branch } x; \\
 \text{not\_shift\_frame } x; \\
 pc(s) < \text{length}(\text{method } CFS \ s) \\
 \hline
 \langle CFS, \sigma \rangle \xrightarrow[s]{(s)} \sigma' \equiv \text{exec}(CFS, \sigma) = \sigma'
 \end{array} \quad (6)$$

## 6 A Pre- and Post-Condition Relation for Execution of Bytecode

### 6.1 The triple $\{P\} xs \{Q\}$ Relation

The Relation  $\{P\} xs \{Q\}$  is **True** if, for all classfiles  $CFS$  containing the instruction sequence  $xs$  bounded by the instructions identified by  $s$  and  $f$ , if the condition  $P$  holds



in state  $\sigma$  and  $\langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma'$  then condition  $Q$  holds in state  $\sigma'$ . Also, the program counter in state  $\sigma$  must point at the first instruction of  $xs$  ( $pc(\sigma) = pc(s)$ ) or at a target instruction within  $xs$ ; the program counter in state  $\sigma'$  must point at the instruction in the method code immediately following  $xs$  ( $pc(\sigma') = pc(f) + 1$ ) or at a target instruction in the method (but outside  $xs$ ).

$$p \ [ \ ] \ q \equiv \forall \ CFS \ \sigma \ \sigma'. \ p(\sigma) \longrightarrow q(\sigma') \quad (7)$$

$$\begin{aligned} \{p\}x : xs\{q\} = & \forall \ CFS \ \sigma \ \sigma' \ s \ f, \\ & (\langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma' \wedge fromto \ s \ f = x : xs \wedge \\ & (pc(\sigma) = pc(s) \vee targ) \wedge (pc(\sigma') = pc(f) + 1) \vee targ) \wedge \quad (8) \\ & p(\sigma) \\ & \longrightarrow q(\sigma') \end{aligned}$$

## 6.2 Lemmas for $\{ \}$ Relation

The following lemmas for precondition strengthening (9) and postcondition weakening (10) have been proved for the `triple` relation.

$$\frac{\forall \sigma. P \ \sigma \supset Q \ \sigma; \quad \{R\} \ xs \ \{Q\}}{\{P\} \ xs \ \{Q\}} \quad (9)$$

$$\frac{\forall \sigma. R \ \sigma \supset Q \ \sigma; \quad \{P\} \ xs \ \{R\}}{\{P\} \ xs \ \{Q\}} \quad (10)$$

## 6.3 High Level Rules

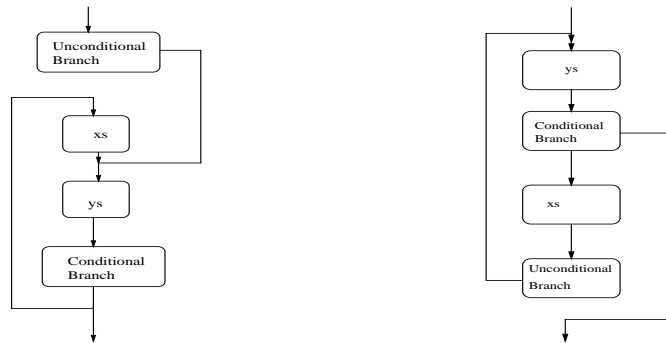
**Imposing Structure—a Rule for Loops in Bytecode** The next stage in developing our Programming Logic, was to develop rules for bytecode patterns corresponding to higher level structures like loops and conditional branches.

```
public class SimpleWhile {
public static void main(String args[])
{
    int i=0;
    while (i<5)
        { i++; }
}
}
```

corresponding to the bytecode

```
Method void main(java.lang.String[])
  0  iconst_0
  1  istore_1
  2  goto 8
  5  iinc 1 1
  8  iload_1
  9  iconst_5
 10  if_icmplt 5
 13  return
```

From examining the bytecode of the example program above and other programs containing `for` and `while` loops, it appeared that there were two equivalent forms that such structures took which seemed to be dependent on the compiler used to produce the bytecode.



where `xs` represents the body of the loop. Having determined this, the next step was to formulate a rule for such bytecode sequences. We decided to begin by considering loops with the structure shown on the left in the diagram above.

The while rule in the standard Hoare logic is

$$\frac{\{P \wedge S\} C \{P\}}{\{P\} \text{ while } S \text{ do } C \{P \wedge \sim S\}} \quad (11)$$

Where  $P$  is an invariant of the loop and  $S$  is the loop guard. In a similar rule for the bytecode representation of a while loop it seemed obvious that `xs` in the diagram above would correspond to  $C$  (the body of the loop), and that the invariant  $P$  did not depend on the language we were dealing with. This left the question of what constituted  $S$ , the loop guard, in the bytecode.

The loop guard is not explicit in the bytecode—as it is in the higher level language. If we consider what role the loop guard plays in the imperative language, however, it becomes

clearer. In the imperative language, evaluation of the loop guard determines whether or not the body of the loop is executed this time round; in the bytecode evaluation of the conditional branch determines whether or not the body of the loop is executed this time round. So this must mean that  $S$  is the condition tested by the conditional branch and our proposed rule looks something like this

$$\frac{\{P \wedge (CBB\_cond)\} \ xs \ \{P\}}{\{P\} [(UBF \ |xs| + 1)]@[xs]@[ys]@[CBB \ |xs@ys|]} \ \{P \wedge \sim (CBB\_cond)\}} \quad (12)$$

But this is not quite accurate. The conditional branch instructions test certain properties of the value or values at the top of the stack, e.g. ‘Jump if the value at top of the stack is equal to Null’, or ‘Jump if the value at the top of the stack is not equal to zero’. A side effect of the comparison is to pop the values involved in this comparison off the stack, with the result that any predicate involving the top of the stack is only correct **immediately before execution of the branch instruction**. So a rule stating that the condition holds anywhere other than just before the branch is executed—including just before execution of  $xs$ , the loop body, as in our proposed rule—is incorrect.

The solution to this particular problem is to ‘wind back’ the conditional being tested until we have a condition in terms of actual variables and values rather than items on the stack. We are, in effect, reconstructing the original guard condition present in the Java source code which is concealed in the bytecode instructions. If we look at the bytecode for the loop we can see that the sequence of instructions  $ys$  is executed prior to the conditional branch every time through the loop. These instructions ‘set up’ the stack so that the correct values are there ready for the comparison. By taking the **weakest precondition** of these instructions and the condition of the branch we are able to determine the actual guard  $S$ .

In the case of our example program, the relevant instructions are

```
16 iload_1
17 iconst_5
18 if_icmplt 5
```

The condition is ‘is the second value on the stack less than the value at the top?’, i.e.  $hd \ (tl \ stk) < hd \ stk$  and we want to determine the weakest precondition of this with respect to the instructions

```
iload_1
iconst_5
```

which, when calculated using the hoisting technique and simplified, is  $lv1 < 5$ . As the value of `i` is stored in local variable 1 (as can be seen from the initialisation instructions at the start of the program

```
0 iconst_0
1 istore_1
```

the weakest precondition is therefore  $i < 5$ , the loop guard of the original Java program, and our rule is now

$$\frac{\{P \wedge wp(ys, CBB\_cond)\} xs \{P\}}{\{P\} [(UBF |xs| + 1)]@[xs]@[ys]@[CBB |xs@ys|] \{P \wedge \sim (wp(ys, CBB\_cond))\}} \quad (13)$$

#### 6.4 Towards a Proof of the Soundness of the Loop Rule

Having proposed this rule for looping patterns of bytecode, we are currently working on a proof of its soundness. The standard Hoare logic while rule for a simple imperative programming language (14) states that if  $P$  is an invariant for one execution of  $C$  whenever  $B$  holds then it is also an invariant for the execution of the statement `while  $B$   $C$` , and that  $B$  will be false on termination of the loop.

$$\frac{\{P \wedge S\} C \{P\}}{\{P\} \text{ while } S \text{ do } C \{P \wedge \sim S\}} \quad (14)$$

Proofs of its soundness can be found in [7] and [17], and they can be broken down into proofs of each of the two properties described above. In [7] these are represented by the following lemmas:

$$\forall C s1 s2. EVAL C s1 s2 \supset (\forall B' C'. (C = \text{while } B' C') \supset \sim (B' s2)) \quad (15)$$

$$\begin{aligned} & \forall C s1 s2. \\ & EVAL C s1 s2 \supset \\ & (\forall B' C'. (C = \text{while } B' C') \supset \\ & (\forall s1 s2. P s1 \wedge B' s1 \wedge EVAL C' s1 s2 \supset P s2) \supset \\ & (P s1 \supset P s2)) \end{aligned} \quad (16)$$

For the Bytecode Programming Logic, equivalent lemmas might be defined as:

$$\begin{aligned} & \forall \sigma \sigma' CFS s f. \langle CFS, \sigma \rangle \xrightarrow[s]{s} \sigma' \wedge \\ & (\text{fromto } s f CFS = \\ & [(UBF |xs| + 1)]@[xs]@[ys]@[CBB |xs@ys|] \supset \\ & \sim ((wp(ys, CBB\_cond)) \sigma')) \end{aligned} \quad (17)$$

$$\begin{aligned}
& \forall \sigma \sigma' CFS \ s \ f. \langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma' \wedge \\
& \text{fromto } s \ f \ CFS = \\
& [(UBF|xs| + 1)]@[xs]@[ys]@[CBB|xs@ys|] \supset \\
& (\forall \sigma \sigma' CFS \ s \ f. \langle CFS, \sigma \rangle \xrightarrow[f]{s} \sigma' \wedge \\
& (\text{fromto } s \ f \ CFS = [xs] \wedge \\
& P \sigma \wedge ((wp(ys, CBB\_cond)) \sigma) \supset P \sigma')) \supset \\
& (P \sigma \supset P \sigma')
\end{aligned} \tag{18}$$

One major difference between the execution of a while loop in the imperative language and a loop sequence in the bytecode is the effect of executing the “structure” of the loop. In the imperative language, the rules for execution state that executing a while statement in an initial state in which the loop guard is false results in an unchanged state. In the bytecode, execution of a loop sequence in which the guard is false in the initial state results in a different state, as evaluating the sequences which constitute the “structure” of the loop means the value of the program counter will have changed. Similarly, with the situation where the body of the loop **is** executed, if a while statement is executed in state  $\sigma$  in which the loop guard is true, we can talk of executing the body of the loop in the same state—evaluation of the loop guard does not affect the state. Again, this is not the case in the bytecode sequence.

This seemed likely to add greatly to the complexity of a proof of the bytecode rule. But closer inspection of the effect of executing the “structure” of a bytecode loop sequence, i.e. the instructions *UBF*, *ys*, and *CBB*, revealed that the only element of the state affected (assuming the instructions concerned satisfied certain constraints) was the program counter. This led us to the idea of **meta-equality** of states: states which are the same apart from the value of the program counter. As the branch condition does not mention the program counter, and assuming that the loop invariant does not either, meta-equal states in the bytecode execution can take the place of equal states in the imperative execution. This definition reduces the differences between the two sets of lemmas and hopefully will simplify the proof.

At present we do not have a proof of the bytecode loop rule, but are considering various proof strategies including structural induction on *xs*, the body of the loop.

## 7 Applications

Once our definition of the Bytecode Programming Logic is complete, we hope to use it in a number of areas affecting JIT compilers.

**Array Bounds Checking** In some JVM implementations the number of native code instructions for the `aload` instruction (load a value from an array onto the stack) [9]

could be halved by the removal of array bounds checking instructions. If it could be proved prior to loading that runtime checks on the array bounds were unnecessary, it would enable standard `aload` instructions to be replaced by faster `aload_quick` instructions—currently not available in any JVM implementation, but obviously advantageous. This tied in with the AJIT project, which also addressed the problem of array bounds checking. We aim to use the Bytecode Programming Logic to prove that, just prior to an array access in a small Java bytecode program, the array reference is non-null and the array index is inside the bounds of the array.

**Exception handling** One other possible optimization involves proving that a method never calls a particular exception, thereby allowing elimination of the code required to handle the exception. Exception handler code accounts for a not insignificant percentage of the code of a method and it is most likely that work will focus on user-defined and thrown exceptions, although several of the standard Java exceptions should be amenable to this treatment (in particular, those concerned with array bounds checking should be largely dealt with by the work of the previous section).

**Virtual Registers** One feature of Java which contributes substantially to its inefficiency is the stack-based model. This does not map directly onto any of the CPUs currently in use, all of which rely on the use of registers, caches and various instruction scheduling algorithms to achieve high performance. Much of the JVM's execution time is wasted on moving values on and off the stack (some estimates put stack operations at 40 per cent of all instructions executed [16]). The AJIT project has developed annotations allowing what they term 'virtual register allocation', calculated at compile time, to be used by the AJIT at runtime to allocate real machine registers. Again, they do not yet formally verify that these allocations are safe, but using the Bytecode Programming Logic it may be possible to do this.

**Memory Disambiguation** Another possible area of interest is memory disambiguation. Proving that two memory operations have no overlapping range—e.g. instruction A writes to a memory location which is subsequently read by instruction B—can lead to further optimizations. Again, this is an area which has been addressed by the AJIT project but without any formal verification as yet.

## 8 Conclusions and Further Work

We have described the initial stages of development for a Programming Logic for Java bytecode programs. This has presented some interesting challenges due to the essential differences between bytecode and the imperative languages: not only must we deal with the problem of `goto` instructions, but much of the control information explicit in a high-level language does not exist in an easily recognised form in the bytecode.

In the future we hope to continue our work on the Logic in order to obtain a more complete system which we can use to prove the properties mentioned in Section 7, leading to faster, more efficient, and trustworthy JIT compilers.

## References

1. Java Card Technology. <http://java.sun.com/products/javacard/>.
2. The LOOP Project. <http://www.cs.kun.nl/~bart/LOOP/index.html>.
3. Project Bali. <http://www4.informatik.tu-muenchen.de/~isabelle/bali/>.
4. VerifiCard Project Summary. <http://www.verificard.org/>.
5. Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers: Principles, Techniques and Tools*. Addison Wesley 37, 1986.
6. Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
7. J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report 256, University of Cambridge Computer Laboratory, 1992.
8. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq SRC, 1998.
9. Tim Harris. A just-in-time Java bytecode compiler. CST Part II project dissertation, University of Cambridge, 1997.
10. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
11. Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java Bytecodes in Support of Optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.
12. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
13. George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.
14. Lawrence C. Paulson. Isabelle, a Generic Theorem Prover. *LNCS*, 828, 1994.
15. Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical report, Technische Universität München, June 1998.
16. P. Wayner. Sun Gambles on Java Chips. *BYTE Magazine*, November 1996.
17. Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.