The Functions of Java Bytecode

Mark P. Jones
Languages and Programming, Department of Computer Science
University of Nottingham, Nottingham NG7 2RD, England.
http://www.cs.nott.ac.uk/~mpj/

Abstract

Java bytecode provides a portable representation for programs that allows executable content to be embedded in web pages, transferred across a network, and executed on a remote user's machine. Features like these provide many new opportunities for developers, but special precautions must be taken to protect users from badly-behaved programs, which might otherwise destroy valuable data or compromise their privacy. To avoid such problems, bytecode programs from untrusted sources must be verified before they are used. If a program passes, then it should be well-behaved, and should not be able to subvert the other security mechanisms of the Java platform. However, if a program fails, then it will be rejected. Clearly, to be sure that it is effective, we need a precise way to understand bytecode verification.

This paper describes the main features of a formal specification for Java bytecode that allows us to reason about the correctness of Java implementations, and to guarantee safety properties of verified bytecode. The key to our approach is to model individual bytecode instructions, and their compositions, as appropriately typed functions in a fairly standard functional language. This gives us a flexible way to build up and extend the instruction set. In addition, it enables us to describe bytecode verification as a well-understood form of type inference, which guarantees that execution of verified programs will not "go wrong."

1 Introduction

The development of web browsers has opened the Internet to a wider audience by replacing the cryptic, text-based frontends of earlier tools with an intuitive and appealing, point-and-click interface. Now, designers are discovering new ways to use the technology by adding *executable content* to web pages. This refers to embedded programs or *applets* that are transferred across the network and executed on users' machines when they visit those pages with a suitable browser. Again, the slick interface of the browser hides a lot of complexity, which means that programs might be downloaded and executed without users realizing that this is happening. Users are keen to take advantage of the new functionality that this provides, but some are worried about the security risks. For example, a badly behaved program could, either maliciously or by genuine error, destroy valuable data stored on a user's computer, or compromise their privacy by transmitting personal information across a network connection.

The Java programming language is one of the most widely used tools for creating executable content, which is distributed in a well-defined bytecode format. This gives Java programs a high degree of portability: there are many different systems connected to the Internet, but most provide an implementation of the *Java virtual machine* (JVM) that is needed to load and execute these bytecode programs. The designers of Java also paid a lot of attention to security. For example, the task of downloading bytecode files is delegated to *class loaders* that will, if necessary, use a bytecode *verifier* to identify badly behaved programs, and to reject any code that does not pass verification. Even then, programs are executed under the supervision of a *security manager* that monitors and restricts access to local resources such as a machine's file store, or network connections.

Although there are many facets to Java security, verification plays a central role. In particular, it acts as a first line of defence against badly formed bytecode files that do not satisfy the static constraints of the JVM, and might therefore subvert higher-level security mechanisms. But the use of a verifier also raises some important questions:

- What, exactly, does the verifier do?
- To what extent is it really safe to execute programs that have been accepted by the verifier?
- What advice can we offer to those producing compilers that target the JVM to ensure that the code they generate will be accepted by the verifier?

The verifier used in the Sun implementation of the JVM is documented in two different ways: in the detailed, natural language presentation of the book by Lindholm and Yellin [10]; and, at a much lower level, in the source code of its implementation. Both have been useful in building or improving implementations of the JVM on a wide range of platforms. However, neither is suitable for the formal understanding of verification that is needed to support a careful, mathematical proof of the safety properties that we are interested in. Another problem with these two alternatives is that they are descriptive rather than prescriptive: they provide us with the means to determine whether given programs are valid, but not with a method for constructing valid programs from the ground up.

This paper outlines a different approach to specifying the JVM that tries to avoid the problems described above. The key idea is to view bytecode files as an unusual concrete syntax for programs with the same dynamic semantics, but in a typed functional language. In this setting, verification of Java bytecode corresponds to the process of parsing and type checking these programs. The functional language that we use has a fairly conventional, polymorphic type system that is well understood, with a soundness result that guarantees that the execution of well-typed programs will not "go wrong" [12]. This result carries over to our work to provide guarantees about the safety of executing bytecode programs. This approach also allows for an incremental specification, because verification of individual JVM instructions can be described and understood independently, rather than as part of some larger analysis for the whole machine. This gives a more prescriptive approach that we hope will provide the authors of compilers with more direct ways to check that generated code will satisfy the constraints of the JVM.

The ideas presented in this paper have been used to describe a fairly large portion of the full JVM instruction set, but space prevents us from including detailed coverage of that here. Instead, we make do with just the key ingredients, illustrated by a much-simplified virtual machine, leaving a more complete presentation to subsequent reports.

Also for reasons of space, we will assume some familiarity with the elements of functional programming in languages like Haskell [17]. Because of this, our approach may at first seem less accessible than others that develop the necessary machinery from first principles. However, there are also significant advantages, for example in being able to adopt and adapt some of the relevant ideas and results that have already been established in that setting, and in being able to experiment with and develop the specification interactively using existing interpreters. In a sense, we use the given type system as a general framework for data-flow analysis, which we can hook into instead of having to build everything ourselves. Moreover, once a final design is reached, we can derive specific rules that deal only with bytecode programs by specializing the full type system with respect to the constructs of the bytecode language. As a result, we can offer designers more freedom in constructing a specification, while also providing users with a simplified version that gives only the details they need to understand the semantics of bytecode.

The remaining sections of this paper are as follows. In Section 2, we show how bytecode programs can be described in terms of functions and function composition. Section 3 shows that we can get a more accurate picture of what each instruction does by giving them different types. Section 4 describes a simple example that highlights the role of type inference in detecting badly-behaved programs. Section 5 explains some of the extensions that are used to obtain a more complete treatment of the JVM. Finally, Section 6 concludes with pointers to related work.

2 Composing instructions

Consider the process of compiling programs written in languages like C or Pascal to a given machine language. In the simplest cases, the result would be a sequence of machine instructions, terminated by a *return* instruction that passes some appropriate value back to the program's caller:

$$i_1;\ i_2;\ \ldots;\ i_{n-1};\ i_n;\ \mathit{return}.$$

A standard way to describe the workings of an abstract machine is to treat each instruction as a transition from one machine state to the next. For example, a 'no operation', or *nop* instruction, which has no effect on the machine state, would be represented by the identity function[1]:

$$\mathit{nop} \quad = \quad \backslash st \to st.$$

Some might argue that the *nop* instruction *does* have an effect on the machine state, namely to increment the program counter. But the way that programs are laid out in memory is an implementation detail, and is important only because of the sequencing of instructions that it implies. We therefore abstract away from such issues, choosing instead to build appropriate notions of control flow into the way that we describe instruction sequencing.

In the general case, most instructions can be modelled by functions[2]:

$$i_1,\ i_2,\ \ldots,\ i_{n-1},\ i_n \quad :: \quad St \to St,$$

where St is a type whose values capture the state of the machine at any point during a computation. The *return* instruction will, however, be treated as a special case. At the end of a program, the state is discarded and an appropriate value of some type Ret is returned to the caller. This suggests that we treat *return* as a function:

$$\mathit{return} \quad :: \quad St \to Ret.$$

Now, the meaning of the compiled program, a function of type $St \to Ret$, is just the composition of the functions corresponding to each individual instruction:

$$St \xrightarrow{i_1} St \xrightarrow{i_2} St \longrightarrow \ldots \longrightarrow St \xrightarrow{i_{n-1}} St \xrightarrow{i_n} St \xrightarrow{\mathit{return}} Ret.$$

[1] An expression $\backslash x \to e$ represents the function that takes an argument x and returns the corresponding value e as its result.

[2] We write $e_1,\ \ldots,\ e_n :: t$ to indicate that each of the expressions on the left of the $::$ symbol have type t. A type of the form $a \to b$ represents the set of functions mapping values of type a to values of type b.

With this approach, the semantics of a particular machine can be specified by giving a type St representing machine states, and a state transition function for each of the machine's instructions. This provides a very accurate picture of the machine's dynamic semantics, explaining what will happen when an instruction is executed. However, because almost all instructions have the same type, it provides only a limited static semantics.

3 Choosing the state type

There are at least two problems with the approach described in the previous section. The first is that, if the type St represents the complete state of the machine, then an instruction of type $(St \to St)$ could, potentially, have unrestricted access to the state of the machine, which is not always desirable. For example, we might like to enforce a restriction that prevents the code for a given method from gaining access to the local variables of its caller.

This can be addressed by replacing St with a type that models only the *local* state of a section of code: its parameters, local variables, temporary storage, and so on. Global state can be reintroduced later using different mechanisms (see Section 5.3). In this paper, we will use a local state that consists of a *frame* or register file, used to store local variables, and an *operand stack*, used to store temporary results during calculations. This suggests that we introduce new types, $Frame$ and $Stack$, and model instructions as functions of type[3]:

$$(Frame,\ Stack) \to (Frame,\ Stack).$$

A second, more serious problem is that, if the state of a machine is represented by any fixed type—whether it is St or $(Frame,\ Stack)$—then there is no way to capture important relationships between the start and finish states of any instructions. As far as the type system is concerned, there is no difference between a *nop* instruction and an instruction that adds two numbers: the type does not reflect any preconditions or postconditions that might distinguish them. So the type system will accept arbitrary, but possibly meaningless sequences of instructions.

To see how that can be resolved, let us reconsider the *nop* instruction, which has no preconditions or postconditions. In other words, it can be executed in any start state, and will deliver the same state as its result. In the Haskell type system, we

[3] A type of the form $(a,\ b)$ represents the set of pairs whose first and second components have types a and b, respectively.

can capture this by assigning *nop* a polymorphic type[4]:

$$
\begin{aligned}
nop &:: (f, s) \to (f, s) \\
nop &= \backslash(f, s) \to (f, s)
\end{aligned}
$$

The type signature in the first line gives a static semantics for *nop*, while the definition on the second line provides its dynamic semantics. The difference from what we saw before is that fixed $Frame$ and $Stack$ types have been replaced with type variables f and s, representing an arbitrary frame and stack, respectively. Following the conventions of Haskell, these variables are bound by implicit universal quantifiers; in more formal notation, we might write the type of *nop* as $\forall f . \forall s . (f, s) \to (f, s)$. This type tells us quite a lot about the *nop* instruction, without even looking at the dynamic semantics: given that f and s are arbitrary, the only reasonable implementation is the identity function, which returns its start state as its result. Informal arguments like this can be made more precise using the free theorem for each type [24], but we do not have space to pursue that any further here.

3.1 Representing stacks

Unlike *nop*, most instructions in our machine will use the stack or the frame components of the local state. We will consider each of these in turn.

Stacks are often used in machine language programs to hold temporary values during a computation. A program can allocate as many temporaries as it needs by pushing values onto the stack—and, when they are no longer required, it is just as easy to reclaim that temporary storage by popping values from the stack.

The list type of Haskell provides a simple representation for stacks, but it is not suitable for our work here: while there are no intrinsic limits on the size of stacks that can be built in this way, the type system does insist that all elements have the same type. This is a serious restriction because we can expect to encounter many different types of temporary value during a calculation and we do not want to have a separate stack for each type. Another way to solve this problem would be to introduce a new, universal sum type with one branch for each case, and then use lists of these values to represent stacks. In effect, each element on the stack would be tagged with a representation of its type. This, however, is also undesirable because it does not correspond to the way that real machines work; it is awkward and messy; and it has the potential for runtime type errors.

[4]We follow a convention of using the same name for types and for related values. In the first line, (f, s) denotes the type of pairs whose components have *types* f and s. In the second line, (f, s) denotes a value of that type, and f and s denote the *values* of its components.

We can avoid these problems by using stack types that document the construction of the stack values that they represent. In essence, we will represent stacks by nested sequences of pairs, writing $x \lhd s$ in both value and type expressions to denote the result of pushing x onto s. For example, if we start with an undefined stack, and push first the integer 12, and then the boolean $True$, then the result is a stack:[5]

$$(True \lhd 12 \lhd undefined) :: (Bool \lhd Int \lhd s).$$

A large stack will have a correspondingly large type, but the stack-based operations that we need to work with use only one or two elements at a time, so we can still give quite compact types for these operations. For example, we can define a variety of stack-based machine instructions such as:

- Load a constant value onto the stack:

$$\begin{array}{lcl} ldc & :: & a \to (f,\, s) \to (f,\, a \lhd s) \\ ldc\ a & = & \backslash(f,\, s) \to (f,\, a \lhd s) \end{array}$$

  The *ldc* instruction is our first example of an instruction with an argument and, once again, takes advantage of polymorphism to allow any type of argument. In the real JVM, there are some restrictions on the type of values that can be pushed; this can be captured by using a type class to constrain the variable a.

- Negate the integer value on the top of the stack:

$$\begin{array}{lcl} ineg & :: & (f,\, Int \lhd s) \to (f,\, Int \lhd s) \\ ineg & = & \backslash(f,\, x \lhd s) \to (f,\, (-x) \lhd s) \end{array}$$

- Add two integer values on the top of the stack:

$$\begin{array}{lcl} iadd & :: & (f,\, Int \lhd Int \lhd s) \to (f,\, Int \lhd s) \\ iadd & = & \backslash(f,\, y \lhd x \lhd s) \to (f,\, (x+y) \lhd s) \end{array}$$

Notice once more the role that polymorphism plays in each of these definitions. Any portion of either the stack or frame that does not participate in an instruction is represented by a (universally quantified) type variable. When the instructions are used in a particular context, these variables will be instantiated to appropriate

[5]We assume that $\lhd$ is right associative, so $a \lhd b \lhd c$ means the same as $a \lhd (b \lhd c)$.

types. For example, the two *ldc* instructions in the following short instruction sequence are used at different types:

$$ldc\ 3;\ ineg;\ ldc\ 4;\ ineg;\ iadd;\ ineg.$$

The composition of these instructions has type $(f,\, s) \to (f,\, Int \lhd s)$. In fact, using the definitions given above, we can use simple equational reasoning to show that this sequence of six instructions has the same effect as a single *ldc* 7 instruction, which, not surprisingly, also has the same type.

3.2 Representing frames

Now we turn our attention to frames, which are used to store the values of parameters and local variables within a block of code. In this case, records provide a natural representation in which the different components of varying type can be identified by a name or a label. In what follows, we will use the record system of Gaster and Jones [7, 6], which provides support for polymorphic operations over extensible records. For example, a type of the form $(x :: a \mid f)$ describes a record that has *at least* an x field of type a, together with whatever other fields are represented by the *row variable* f. This notation is particularly useful in work with frames because it allows us to refer to individual components like x, without having to say anything more about the rest of the frame.

The following examples show the kind of instructions that we might use to provide access to frame components.

- To load a value from the field x in the frame, pushing the result onto the top of the stack:

$$\begin{array}{lcl} load\_x & :: & ((x :: a \mid f),\, s) \to ((x :: a \mid f),\, a \lhd s) \\ load\_x & = & \backslash(f,\, s) \to (f,\, (f\,.\,x) \lhd s) \end{array}$$

- To pop a value from the top of the stack and store it in the x component of the frame:

$$\begin{array}{lcl} store\_x & :: & ((x :: b \mid f),\, a \lhd s) \to ((x :: a \mid f),\, s) \\ store\_x & = & \backslash((x = \_ \mid f),\, a \lhd s) \to ((x = a \mid f),\, s) \end{array}$$

To ease the presentation, we have used symbolic names for frame components. In the real JVM, frame components are accessed by position, and it is the task of a compiler to map the variables in a user's program to appropriate slots in a frame. In fact, if two variables have non-overlapping live ranges (so that at most one of the variables is in use at any given time), then a compiler might even use the same frame slot for the two variables. Our approach also permits this, because individual frame components can hold different types of value at different stages of a computation. To see how this can happen, notice that the type for the $store\_x$ instruction does not place any restrictions on the type of value in slot x before the instruction is executed (represented by the type variable b). So, once again, polymorphism plays a key role.

To illustrate the use of these instructions, the following code fragment shows how we might implement an assignment of the form $x = y + z$:

$$load\_y;\ load\_z;\ iadd;\ store\_x\ .$$

It is easy to show that this short sequence of instructions has type:

$$((x :: a,\ y :: Int,\ z :: Int \mid f),\ s) \to ((x :: Int,\ y :: Int,\ z :: Int \mid f),\ s).$$

Again, the occurrences of type variables here tell us quite a lot about the sequence of instructions. For example, although the code fragment uses the stack internally, the fact that it is polymorphic in s tells us that the overall effect is to leave the stack unchanged.

Following the patterns above, we would need to introduce a *load* and *store* instruction for each different name x that we might want to use to label a frame component. This does not cause any technical problems because all frame labels are known at compile-time, but it is rather unwieldy. A more attractive approach is to use first-class labels [7] to define a single *load* and *store* instruction, each parameterized by a frame label.

4 A simple example

To see how the limited instruction set that we have built up in the previous sections might be used in practice, consider the following fragment of a Java program:

```
static int sum(int n) {  // Add up the numbers from 1 to n
    int t = 0;
    for (int c=0; c!=n; c++)
        t = t + c;
    return t;
}
```

Given this as input, a simple Java compiler might generate the following bytecode as its output:

$$\begin{array}{lcl}
entry & = & ldc\ 0;\ store_t;\ ldc\ 0;\ store_c; \\
 & & goto\ test \\
 & & \\
loop & = & load_c;\ load_t;\ iadd;\ store_t; \\
 & & load_c;\ ldc\ 1;\ iadd;\ store_c; \\
 & & goto\ test \\
 & & \\
test & = & load_c;\ load_n;\ ineg;\ iadd; \\
 & & if_ne\ loop\ end \\
 & & \\
end & = & load_t; \\
 & & ireturn
\end{array}$$

This particular fragment of code breaks down into four basic blocks, represented here by the four values *entry*, *loop*, *test* and *end*. In particular, *entry* represents the entry point of the method, for which we assume that a frame has been created to contain storage for the parameter n, and for the (uninitialized) local variables c and t. This program uses three new instructions for describing control flow in bytecode programs:

- The *ireturn* instruction is used to return an integer value, which it expects to find on the top of the stack. The frame, and any other values on the method's stack are discarded when it returns:

$$\begin{array}{lcl}
ireturn & :: & (f,\ Int \lhd s) \to Int \\
ireturn & = & \backslash(f,\ n \lhd s) \to n
\end{array}$$

- The *goto* instruction causes an unconditional branch to another section of code. In fact, *goto* is just the identity function on values of type $((f,\ s) \to a)$, which is the type of a label in our current framework:

$$\begin{array}{lcl}
goto & :: & ((f,\ s) \to a) \to ((f,\ s) \to a) \\
goto\ lab & = & lab
\end{array}$$

- The *if_ne* instruction is used to implement a conditional branch: if the integer on the top of the stack is non-zero, then it causes a branch to the first label, otherwise it branches to the second label. In either case, the integer value is removed from the top of the stack:

$$\begin{array}{lcl}
if_ne & :: & ((f,\ s) \to a) \to ((f,\ s) \to a) \to (f,\ Int \lhd s) \to a \\
if_ne\ l\ m & = & \backslash(f,\ n \lhd s) \to \mathbf{if}\ n \neq 0\ \mathbf{then}\ l\ (f,\ s) \\
 & & \qquad\qquad\qquad\qquad\quad \mathbf{else}\ m\ (f,\ s)
\end{array}$$

In the real JVM, the if_ne instruction has only one argument, leaving execution to continue at the next instruction in memory if the value on the top of the stack is zero. However, the next instruction will always be at the beginning of another basic block, so we have chosen to simplify the presentation by making both labels explicit. An alternative would have been to use a continuation passing encoding for instructions.

Given the goals of this work, it is particularly interesting to look at the types for each of the four labels:[6]

$$\begin{array}{lll}
entry & :: & ((n :: Int,\ c :: None,\ t :: None),\ s) \rightarrow Int \\
loop & :: & ((n :: Int,\ c :: Int,\ t :: Int),\ s) \rightarrow Int \\
test & :: & ((n :: Int,\ c :: Int,\ t :: Int),\ s) \rightarrow Int \\
end & :: & ((n :: a,\ c :: b,\ t :: Int),\ s) \rightarrow Int
\end{array}$$

The $None$ type that we have used in the type of $entry$ has only one value, $\perp$, which indicates an $undefined$ element. It is used here to reflect the fact that the c and t components of the frame have not been properly initialized at the beginning of the compiled code. On the other hand, the code for *end* does not make use of the n or c components of the frame, and this is reflected in the type for *end*, which allows arbitrary types of value in those slots. In this way, the type system is providing a kind of live variable analysis.

In most cases, adding or removing instructions in the program above—and so simulating the possible corruption of a bytecode as it travels across a network connection—will prevent the resulting program from type checking. For example, if the two instructions *ldc* 0; *store_t* were removed from the code for $entry$, then the code for *test* would be rejected: for the use of *iadd* to be valid, the t slot must have been previously initialized with an integer value. Of course, this is exactly the kind of badly behaved program—in this case, attempting to use an uninitialized variable—that we want the verification process to detect.

On the other hand, the program will still type check if the *ineg* instruction is removed from the code for *test*. The resulting program would no longer function correctly (i.e., as the person who wrote the Java definition of `sum` would expect) but it is not badly behaved, and so would still be expected to pass the verification process. Verification can provide guarantees that programs are well-behaved, but it will never be able to guarantee anything more about what those programs might actually do.

[6] Automatic type inference will result in more general types than the ones shown here. For example, they will allow the code to be executed in frames that include slots for other variables, not just n, c, and t. We use more restricted types to simplify the presentation.

5 Extending the model

The ideas that we have described in the previous sections are simple and elegant, but real Java programs are much more complicated than the examples that we have seen so far and there are many other bytecodes that we need to specify to give a complete treatment of the JVM. In this section, we show how our basic model extends smoothly to deal with some of these features.

5.1 Handling multi-word values

The approach that we have used so far suggests that any type of value can be stored in a single JVM stack or frame entry. However, while small values, like integers and characters, can be held in a single word, larger values must be implemented either as boxed values (i.e., as word-sized pointers to larger values stored in the heap) or by using multiple words. In the JVM, the first approach is used to deal with objects; in fact that is the only option because objects contain mutable state, and may be shared. The second approach is used, for example, to deal with values of the primitive type `long`, each of which takes two words of storage.

To model this, we introduce two types $Long_1$ and $Long_2$ whose values are the word-sized results that we get when we break a *Long* value into its two constituent pieces:

$$\begin{array}{lcl} breakLong & :: & Long \to (Long_1,\, Long_2) \\ makeLong & :: & (Long_1,\, Long_2) \to Long \end{array}$$

The JVM specification leaves all decisions about the conversion between *Long* values and pairs of words to individual implementations. We will therefore treat $Long_1$ and $Long_2$ as abstract datatypes, and will assume that *breakLong* and *makeLong* are mutual inverses. Given this framework, we can define long analogues for each of the integer instructions described previously.

$$\begin{array}{lcl} lneg & :: & (f,\, Long_1 \lhd Long_2 \lhd s) \to (f,\, Long_1 \lhd Long_2 \lhd s) \\ lneg & = & \backslash(f,\, x \lhd y \lhd s) \to \mathbf{let}\ (u,\, v) \;\; = \;\; breakLong\ (-makeLong\ (x,\, y)) \\ & & \qquad\qquad\qquad\qquad\;\; \mathbf{in}\ (f,\, u \lhd v \lhd s) \end{array}$$

The reason we use a different abstract datatype for each half of a *Long* value becomes more apparent when we consider some of the JVM's instructions for stack manipulation:

$$\begin{array}{lcl} pop & :: & (Word\ a) \Rightarrow (f,\, a \lhd s) \to (f,\, s) \\ pop2 & :: & (TwoWord\ a\ b) \Rightarrow (f,\, a \lhd b \lhd s) \to (f,\, s) \end{array}$$

In this case, type class constraints are used to restrict the use of polymorphism and so ensure that stack integrity is preserved. In particular, this means that JVM bytecode is not allowed to treat the two parts of a `long` value as independent entities. Thus the $(Word\ a)$ constraint is defined to hold only for types a that fit in a single word and represent a complete value[7].

$$Word\ a \Leftrightarrow (a = Int) \vee (a = Boolean) \vee (a = Float) \vee \ldots$$

On the other hand, the *pop*2 instruction can be used to remove two word values from the stack, which may either be two separate one word value, or the two parts of a single two word value. Thus the $(TwoWord\ a\ b)$ is defined to satisfy:

$$TwoWord\ a\ b \Leftrightarrow (Word\ a \wedge Word\ b) \vee (a = Long_1 \wedge b = Long_2) \vee \ldots$$

5.2 Dealing with effects

Of course, real Java programs depend on features like exceptions, I/O, and mutable state in ways that our simple, functional model may not seem to support. But, in fact, these features *can* be accommodated, simply by changing the way that we interpret the $\rightarrow$ symbol. More precisely, in assigning types to instructions, we just need to reinterpret function types like $a \rightarrow b$ as types of the form $a \rightarrow m\ b$. The symbol m used here represents a *monad* that encapsulates the specific computational features that we are interested in [13, 25]. To explore this idea in detail would take us beyond the scope of the current paper, so this section aims only to explain the basic concepts.

For the purposes of this paper, it is enough to think of a monad as a particular kind of parameterized datatype with the intuition that, if m is a monad, then the type $m\ a$ represents *programs* that return *values* of type a. The distinction between programs and values is critical: for example, the fact that a method in a Java program returns an integer result does not tell us whether the body of that method makes use of side-effects, exceptions, or other features. In fact, with the approach described here, a method will be represented by a function with a type of the form $p \rightarrow m\ r$: in words, a method is a function taking parameters of type p and producing a program that delivers results of type r.

Every monad comes with two important operations:

- A *unit* operator: $unit :: a \rightarrow m\ a$. Intuitively, $unit\ x$ is just a program that returns the value x, without using any of the computational features provided by m. So *unit* is just like an identity function, and corresponds to the *nop* instruction.

[7] In practice, these constraints are defined by appropriate collections of instance declarations

- A composition operator: $(;) :: (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$. To understand this, think of $(f;\ g)$ as a function that takes an argument x of type a, runs the program $(f\ x)$ to obtain a result y of type b, and then runs $(g\ y)$ to obtain a result of type c. In this paper, we will use the composition operator to combine the effects of individual instructions, and so to give a meaning to complete sequences of code.

Of course, these two operators were just the starting point for our presentation of the simplified JVM in Section 2. To complete the definition of a monad, we need these operators to satisfy three simple laws:

$$
\begin{array}{lcl}
f;\ \mathit{unit} & = & f \\
\mathit{unit};\ g & = & g \\
(f;\ g);\ h & = & f;\ (g;\ h)
\end{array}
$$

The first and second laws here show that *unit* is both a right and a left identity for the composition operator, confirming the earlier suggestion that *unit* behaves much like a *nop* instruction. The third law tells us that composition is associative, which means that we can write down sequences of instructions $i_1;\ i_2;\ \ldots;\ i_{n-1};\ i_n$ without worrying about the use of parentheses.

There are many different monads that we might be interested in using, and, in Haskell, we can capture the key properties from our description of monads by defining a class:[8]

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{unit} & :: & a \to m\ a \\
(;) & :: & (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)
\end{array}
\end{array}
$$

The instructions that we have defined so far can be modified to fit the monadic framework by adding a reference to the monad in each types, and by inserting appropriate calls to *unit* in each implementation:

$$
\begin{array}{lcl}
\mathit{nop} & :: & (\mathit{Monad}\ m) \Rightarrow (f,\ s) \to m\ (f,\ s) \\
\mathit{nop} & = & \backslash(f,\ s) \to \mathit{unit}\ (f,\ s)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ldc} & :: & (\mathit{Monad}\ m) \Rightarrow a \to (f,\ s) \to m\ (f,\ a \lhd s) \\
\mathit{ldc}\ a & = & \backslash(f,\ s) \to \mathit{unit}\ (f,\ a \lhd s)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ineg} & :: & (\mathit{Monad}\ m) \Rightarrow (f,\ \mathit{Int} \lhd s) \to m\ (f,\ \mathit{Int} \lhd s) \\
\mathit{ineg} & = & \backslash(f,\ x \lhd s) \to \mathit{unit}\ (f,\ (-x) \lhd s)
\end{array}
$$

[8] In fact, Haskell already defines a class of monads using a rather different formulation to the one that we have adopted here, although the underlying structure is essentially the same.

$$\begin{array}{lcl} iadd & :: & (Monad\ m) \Rightarrow (f,\ Int \lhd Int \lhd s) \rightarrow m\ (f,\ Int \lhd s) \\ iadd & = & \backslash(f,\ y \lhd x \lhd s) \rightarrow unit\ (f,\ (x+y) \lhd s) \end{array}$$

$etc\ldots$

In retrospect, it is unfortunate that we did not write our definitions in this manner to begin with. However, now that we know what is required, it is easy to ensure that any new instructions we add are defined in this style. Moreover, once this change has been made, it is easy for us to use different underlying monads, without any further need to rewrite the definitions for each instruction.

None of the instructions described above have any real computational effect. This is reflected by the fact that their types are polymorphic in the monad m. It is more interesting to look at instructions that do have computational effects. A simple example of this is the *idiv* instruction:

$$idiv\ \ ::\ \ (ThrowsMonad\ m) \Rightarrow (f,\ Int \lhd Int \lhd s) \rightarrow m\ (f,\ Int \lhd s)$$

Notice that this is very much like the type that we saw for *iadd* except that the ($Monad\ m$) constraint has been replaced by ($ThrowsMonad\ m$); unlike *iadd*, the *idiv* instruction will throw a *ArithmeticException* if the divisor is zero. The *ThrowsMonad* class used here is a subclass of *Monad*, which means that m must support all the operations that a monad supports, together with any of the extra operations that are required for throwing an exception[9]

5.3 Representing objects and global state

Another common application of monads is to encapsulate and describe computations using global state. In fact, for the purposes of the JVM, it is quite useful to define a whole family of monads that divide up the state into (overlapping) regions to capture and enforce the access restrictions associated with individual class members. To illustrate this, we first need to say a little about the way that objects are represented. Broadly speaking, for each Java class X, we define a type XT and a (Haskell-style) type class XC. The type XT represents concrete instances of C, while the type class XC represents the subclasses of X (including

[9] Our discussion of exception handling has been simplified in two significant ways for the purposes of presentation. First, we have grouped all exceptions together in a single *ThrowsMonad* class instead of providing a separate class for each different exception type. Second, real JVM bytecode does not guarantee structured (i.e., nested) exception handlers, so the real *idiv* instruction requires an extra parameter for its exception handler.

XT). For example, a class definition

```
class B extends A { ... }
```

will be mapped to a collection of type and class definitions:

$$\begin{array}{ll}
\textbf{data}\ BT\ \ =\ \ \ldots & \\
\textbf{class}\ AC\ o \Rightarrow BC\ o\ \textbf{where} & \ldots \\
\textbf{instance}\ AC\ BT\ \textbf{where} & \ldots \\
\textbf{instance}\ BC\ BT\ \textbf{where} & \ldots
\end{array}$$

The first line here defines the concrete representation for objects of class B. The second line reflects the inheritance relation between the classes A and B: the variable o represents an arbitrary instance of the class, and the definition specifies that, if o is a subclass of B, then it must also be a subclass of A. The third line is an instance declaration that specifies the definitions for any fields and methods that class B inherits from A, while the fourth line specifies any new fields or methods that B provides.

Now consider the following program fragment:

```
package P;

class A {
  int x;
  public boolean y;
  static float z;
  ...
}
```

From this definition, we can see that every object of class `A` has its own `x` and `y` fields, so we would expect a corresponding type class of the form:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ AC\ o\ \textbf{where}} \\
\quad A.x & :: & PackageP\ m \Rightarrow Field\ m\ o\ Int \\
\quad A.y & :: & ClassP.A\ m \Rightarrow Field\ m\ o\ Boolean \\
\quad & \ldots &
\end{array}$$

To avoid unwanted name clashes, we qualify the name of each field with the name of the class in which the field occurs. Both $A.x$ and $A.y$ have types of the form $Field\ m\ o\ t$ describing a field of type t in an object of type o, accessed in a monad m. (The $Field$ datatype and the $Static$ datatype used below are predefined types, much like Int or $Float$.) The two fields differ in the constraints that they place on the monad m. In particular, because x has package scope, it can only be accessed from within package `P`, which is captured by the constraint $(PackageP\ m)$. On the other hand, $A.y$ has public scope and can be used anywhere that class `A` is in scope.

The static field `z` in class `A` is mapped to a global value:

$$A.z \ :: \ PackageP\ m \Rightarrow Static\ m\ Float$$

Once again, we can see that $A.z$ has package access, as reflected by the constraint on m. This time, however, we use a type of the form $Static\ m\ t$, representing a field of type t, accessed in a monad m. Static fields are not associated with any particular object, and hence there is no place for an object type o in this case.

The JVM provides separate instructions for accessing the fields of a class:

$$\begin{aligned} getstatic &:: Monad\ m \Rightarrow Static\ m\ a \rightarrow (f,\ s) \rightarrow m\ (f,\ a \lhd s) \\ getfield &:: Monad\ m \Rightarrow Field\ m\ o\ a \rightarrow (f,\ o \lhd s) \rightarrow m\ (f,\ a \lhd s) \end{aligned}$$

The types of these instructions are similar to those of the *load* operations in Section 3.2, except that they are parameterized by appropriate fields specifications. Note also that the $getfield$ instruction expects an object of type o on the top of the stack, which it replaces with the contents of the selected field.

Judging from the $(Monad\ m)$ constraint that appears in the types of both $getstatic$ and $getfield$, one might be tempted to think that these instructions have no computational effect. Indeed, that is quite possible if the fields that they refer to have fixed, constant values. However, in general, additional constraints on the monad m are introduced when the instructions are applied to a particular field. For example, to load the contents of the $A.z$ field onto the top of the stack we use the instruction:

$$getstatic\ A.Z \ :: \ PackageP\ m \Rightarrow (f,\ s) \rightarrow m\ (f,\ Int \lhd s).$$

The constraint inferred here shows that this instruction can only be used in a method for a class in package `P`.

6 Related work

There has been a lot of work to investigate the security aspects of Java. Much of this is available on the World-Wide Web, and there are several useful lists of pointers to these resources such as the official list at Sun Microsystems [22], and the Java Security Hotlist at Reliable Software Technologies [5]. A recently published book by McGraw and Felten provides accessible and clear overviews and explanations of the Java security model [11].

The work presented in this paper is directed at the process of verification, which is just one part of Java security. Members of the Secure Internet Programming group at Princeton [8] have worked on this and other areas such as extensible security architectures [26] and the security of static typing with dynamic linking [2]. Even with a formal specification of bytecode verification, it is still possible for errors or design flaws in other parts of the system, particularly the standard APIs, to compromise security or type safety. For example, recent work by Khurshid that demonstrates problems with binary compatibility [9], while Saraswat has highlighted problems with class loaders [19].

At the University of Washington, Project Kimera is developing a new security model for Java. As part of this work, they have identified a collection of nearly 600 axioms in the JVM specification, and they have used these to build a secure verifier. An automated testing process, which generates many random mutations of bytecode files, was used to test the Kimera verifier, and to uncover numerous flaws in other Java implementations [20].

Recent work by Stata and Abadi [21], and by Freund and Mitchell[4] has similar goals to us in building a formal model of Java bytecode instructions. The former focuses on bytecode subroutines, while the latter deals with object initialization, neither of which are discussed in this paper. There has also been similar work by Qian [18], and by Morrisett *et al* in the more general setting of typed assembly language [15, 14]. Unlike our approach, these authors have developed static and dynamic semantics for bytecode from first principles. As a result, their work may be more immediately accessible to members of the Java community who are not familiar with functional programming. On the other hand, with our approach, there is an opportunity to exploit established results and ideas from functional programming, and to experiment with the specification as it evolves using existing functional language implementations.

Another formal model of the JVM has been developed by Computational Logic, Inc. using ACL2, a mathematical logic based on Common Lisp [1]. The result is intended to serve as a basis for rigorous, formal analysis of the JVM. Their model is called the "defensive" JVM because it includes run-time checks to detect and prevent unsafe execution.

Questions of type safety for the Java language itself have been investigated by Drossopoulou and Eisenbach [3], Syme [23], and Nipkow and von Oheimb [16], taking advantage of automatic theorem provers to establish useful soundness results. These results do not apply directly to the JVM because there is no guarantee that bytecode files will be generated by type-preserving compilers from Java source. One interesting idea, attributed to Andrew Appel [11, Page 129], is to verify bytecode files by first *decompiling* them into Java source, and then recompiling them with a trusted compiler.

Acknowledgements

Thanks to members of the Languages and Programming group at Nottingham, particularly Benedict Gaster and Claus Reinke, and also to Phillip Yelland, for their interest and encouragement during the long gestation of this paper.

References

[1] Richard M. Cohen. The defensive Java virtual machine, version 0.5 alpha release, May 1997. Available from `http://www.cli.com/software/djvm/`.

[2] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, April 1997.

[3] Sophia Drossopoulou and Susan Eisenbach. Java is type safe—probably. In *The 11th European Conference on Object Oriented Programming*, June 1997.

[4] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, October 1998.

[5] Reliable Software Technologies Gary McGraw. The Java security hotlist. Available from `http://www.rstcorp.com/javasecurity/links.html`.

[6] Benedict R. Gaster. Polymorphic extensible records for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997.

[7] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[8] Princeton University Secure Internet Programming group. Home page at: `http://www.cs.princeton.edu/sip/`.

[9] Sarafraz Khurshid. Binary compatibility and type soundness of Java, 1997. Available from `http://www-ala.doc.ic.ac.uk/~ scd/sk.html`.

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

[11] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1997.

[12] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[13] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[14] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[15] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *Conference record of POPL '98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, January 1998.

[16] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Conference record of POPL '98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170, San Diego, CA, January 1998.

[17] John Peterson and Kevin Hammond. Report on the programming language Haskell, a non-strict, purely functional language (version 1.4). Technical report, April 1997. Available from `http://www.haskell.org`.

[18] Zhenyu Qian. A formal specification of Java(tm) virtual machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java(tm)*. Springer Verlag LNCS, 1998.

[19] Vijay Saraswat. Java is not type safe. Technical report, AT&T Research, August 1997. Available from `http://www.research.att.com/~ vj/bug.html`.

[20] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. Security flaws in Java implementations. Technical report, Project Kimera, University of Washington, 1997. Available from `http://kimera.cs.washington.edu/flaws/index.html`.

[21] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference record of POPL '98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, CA, January 1998.

[22] Sun Microsystems, Inc. Java security. `http://java.sun.com/security/`.

[23] Don Syme. Proving Java type soundness. Technical report 427, Computer Laboratory, University of Cambridge, June 1997.

[24] Philip Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89*, Imperial College, London, September 1989.

[25] Philip Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.

[26] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. Technical Report 546-97, Department of Computer Science, Princeton University, April 1997.