A Simple, Comprehensive Type System for Java Bytecode Subroutines

Robert O'Callahan
Carnegie Mellon University
500 Forbes Avenue
Pittsburgh, PA
1 412 2685728
roc+@cs.cmu.edu

ABSTRACT

A type system with proven soundness is a prerequisite for the safe and efficient execution of Java bytecode programs. So far, efforts to construct such a type system have followed a "forward dataflow" approach, in the spirit of the original Java Virtual Machine bytecode verifier. We present an alternative type system, based on conventional ideas of type constraints, polymorphic recursion and continuations. We compare it to Stata and Abadi's JVML-0 type system for bytecode subroutines, and demonstrate that our system is simpler and able to type strictly more programs, including code that could be produced by Java compilers and cannot be typed in JVML-0. Examination of real Java programs shows that such code is rare but does occur. We explain some of the apparently arbitrary constraints imposed by previous approaches by showing that they are consequences of our simpler type rules, or actually unnecessary.

Keywords

Java, bytecode, types, subroutines, continuations, polymorphic recursion.

1. INTRODUCTION

The Java bytecode language (referred to hereafter as JVML), as implemented by the Java Virtual Machine [5], has become a widely used medium for distributing platform-independent programs. It attempts to enforce safety properties for these programs, using a mixture of static and dynamic checking. This is particularly important because the programs may be untrusted. For example, if a program can access memory in undisciplined ways, then it may be able to steal private information.

However, people [e.g., 1] have found a number of weaknesses in the JVM's checks that have caused it to accept programs with unsafe and possibly insecure behavior. Therefore it seems desirable to formalize the Java bytecode language and its type system and to prove safety. Researchers [8, 7, 2] have formulated type systems for subsets of the Java bytecode language and proven their soundness.

Copyright © 1998 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

The goal of these alternative systems was to formalize, explain and correct the JVM verifier. Thus they are similar to the JVM verifier in spirit. However, we found it instructive to construct a type system from scratch based on conventional ideas such as continuations and polymorphic recursion, and compare it to the JVM verifier and the aforementioned type systems. To facilitate direct comparison, we have constructed a language JVML-0-C which is practically identical to Stata and Abadi's JVML-0 in syntax and dynamic semantics, but which has an entirely different static semantics. JVML-0-C satisfies the same soundness theorem as JVML-0.

The type system of JVML-0-C is similar to the Stack-based Typed Assembly Language of Morrisett et al. [6] (hereafter referred to as "STAL"). By targeting the JVML-0 bytecode language, we demonstrate that this approach leads to type rules that are somewhat simpler than the corresponding rules of the JVM-like systems. Furthermore, JVML-0-C is able to type strictly more programs. In particular, it admits programs that perform "non-local returns", which occur when a bytecode subroutine returns to code that was not its immediate caller, skipping some frames on the hypothetical subroutine call stack. Such programs are not typable in JVML-0, but can be accepted by the JVM. This is important because such code can be generated by the Java compiler. As we describe below, examples are very rare but do occur in real Java code. The difficulty of typing these programs is due to JVML-0's need to label instructions with the subroutine they belong to, which is not required in JVML-0-C.

A significant benefit of studying JVML-0-C is that it illuminates some of the design choices made in the JVM-like type systems. Hitherto, the type constraints in those systems have been evaluated according to three criteria: they match the "expected" behavior of bytecode programs, they are together sufficient to prove soundness, and they admit code generated by the Java compiler. This gives great latitude in designing the constraints, but this latitude can obscure the motivation and consequences of the design choices. In contrast, each type constraint in JVML-0-C has its own individual, clear rationale. We therefore use JVML-0-C as a basis for evaluating the rules of JVML-0 (and by analogy, the rules of other JVM-like type systems):

- Some properties described by others as "needed for type safety", such as a LIFO discipline for subroutine calls, can in fact be violated by well-typed JVML-0-C programs, and thus are not fundamentally necessary for type safety.

instruction	::=	`inc`	increment the integer on top of the stack
	|	`pop`	discard the top value from the stack
	|	`push0`	push the integer zero onto the stack
	|	`load` *x*	push the value of variable *x* onto the stack
	|	`store` *x*	pop from the stack and store the value into variable *x*
	|	`if` *L*	pop from the stack and branch to L if non-zero
	|	`jsr` *L*	jump to L, push the next instruction address onto the stack
	|	`ret` *x*	jump to the address in variable *x*
	|	`halt`	stop, returning the integer on top of the stack

Figure 1: JVML-0/JVML-0-C Syntax

- JVML-0 is more restrictive than necessary in several other ways. In particular, the need to label instructions with the subroutine they belong to is a key source of complexity and inflexibility, and is done away with in JVML-0-C by typing return addresses as continuations (hence the "C" in "JVML-0-C").
- An interesting property enforced by JVM-like systems is that a return address can only be "used" once. It turns out that this property is a direct consequence of disallowing recursive return address types in JVML-0-C.
- Most of the JVML-0 constraints are implied by the JVML-0-C constraints, and therefore obtain the same justifications.

2. OVERVIEW OF THE LANGUAGE AND TYPE SYSTEM

2.1 The language

Except for the type system, our language JVML-0-C is almost identical to JVML-0 — a bytecode-like language with an operand stack, simple arithmetic, conditional branches, local variable loads and stores, and subroutine calls and returns (see Figure 1). A program includes the code for only a single method — there are no objects, nor a heap. The syntax and dynamic semantics are unchanged. We prove the same soundness theorem as for JVML-0, slightly strengthened to guarantee that an integer is returned.[1]

2.2 JVML-0-C types

As in JVML-0, a type is an abstraction of the state on entry to an instruction. We define state types, stack types and local variable types that are very similar to JVML-0. The key differences are the handling of return address types and the introduction of type variables into JVML-0-C.

The program type P is a map from instruction addresses to state types.

A *state-type* Σ is a pair of a stack-type θ and a locals-type T. It gives the types that hold on entry to a bytecode instruction.

Σ ::= (θ, T)

A *stack-type* θ is either a stack type variable ϕ, an application of a constructor taking the type of the top of the stack and the type of the rest of the stack, or NIL, the type of the empty stack. As in STAL, the introduction of polymorphic stack type variables enables recursive subroutines. It's also necessary for translating JVML-0 types to JVML-0-C types; see Section 9.

θ ::= $\tau \cdot \theta$ | ϕ | NIL

A *locals-type* T is a finite map from local variables to their types.

T : VAR $\rightarrow$ *element-type*

The type τ of an element of the stack or of a local variable (an *element-type*) is one of the integer type INT, the type of all values TOP, a type of a return address (continuation) "Cont", or an element type variable α.

A return address type gives a state-type Σ that must hold for the program state whenever the return address is returned to. This allows us to write rules for the subroutine call and return instructions that are "local", referring only to the types of the caller instruction, the called instruction, and the returned-to instruction. We also avoid any need to explicitly name subroutines or label instructions as belonging to particular subroutines. In addition, the combination of type variables with continuations allows a flexible polymorphism in subroutines in a natural way.

τ ::= INT | TOP | Cont(Σ) | α

The scope of a type variable is the instruction type in which it appears. In effect, the type of an instruction universally quantifies over all the type variables that appear within it. In the type rules, whenever we refer to the type of an instruction, we instantiate it with fresh bindings for the type variables.

The type rules are given in Section 3.

2.3 Example

Figure 2 is an example of a JVML-0-C program with a valid type assignment. It pushes zero and an unusable value onto the stack, calls a subroutine to swap them, and then halts with the zero on top of the stack. The subroutine is polymorphic in the types of the stack values that it swaps. This kind of polymorphism is not available in either the JVM or JVML-0; subroutines in these

[1] We require methods to return integers, whereas JVML-0 leaves the return type unspecified.

i	$P[i]$	Δ_i
0	push0	(NIL, [0: α_0, 1: α_1, 2: α_2, 3: α_3])
1	load 0	(INT · NIL, [0: α_0, 1: α_1, 2: α_2, 3: α_3])
2	jsr 4	(α_0 · INT · NIL, [0: α_0, 1: α_1, 2: α_2, 3: α_3])
3	halt	(INT · α_0 · NIL, [0: α_0, 1: α_1, 2: α_0, 3: INT])
4	store 1	(Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]) · α_4 · α_5 · NIL, [0: α_0, 1: α_1, 2: α_2, 3: α_3])
5	store 2	(α_4 · α_5 · NIL, [0: α_0, 1: Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]), 2: α_2, 3: α_3])
6	store 3	(α_5 · NIL, [0: α_0, 1: Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]), 2: α_4, 3: α_3])
7	load 2	(NIL, [0: α_0, 1: Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]), 2: α_4, 3: α_5])
8	load 3	(α_4 · NIL, [0: α_0, 1: Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]), 2: α_4, 3: α_5])
9	ret 1	(α_5 · α_4 · NIL, [0: α_0, 1: Cont(α_5 · α_4 · NIL, [0: α_0, 1: TOP, 2: α_4, 3: α_5]), 2: α_4, 3: α_5])

Figure 2: A polymorphic swap subroutine

systems are only polymorphic over local variables that they do not touch in any way.

The type for instruction 0 indicates that on entry, it expects an empty stack, and the local variables can each have any type.

The type for instruction 4 indicates that on entry, the local variables can have any type, but the stack must contain three elements. The top element must be a return address, the next element has some arbitrary type α_4, and the last element has some arbitrary type α_5. The returned-to instruction must accept a state where the stack has two elements, the top having type α_5 and the next having type α_4. It must also accept local variable 0 having the same type it has on entry (α_0), local variable 1 being unusable (TOP), local variable 2 having type α_4 and local variable 3 having type α_5. (Local variable 1 becomes unusable on return because it holds the return address when the subroutine returns at instruction 9; see Section 3.4 for why this is so.)

Note that instruction 3 is a valid return address for the subroutine because its type "matches" the return address type. In type for instruction 3, α_1 can be any type, so we can substitute TOP for it. Similarly, we can substitute α_0 for α_4, and INT for α_5. The resulting "instantiation" is precisely the return address type, which shows that whatever state the subroutine returns in will be acceptable at instruction 3.

2.4 Overview of the rest of the paper

Section 3 gives the type rules for JVML-0-C.

Section 4 compares JVML-0-C with JVML-0 and the JVM. In Sections 5.1 and 5.2 we discuss how JVML-0-C admits certain programs that are accepted by the JVM but not by JVML-0. In Section 5.3 we mention some features that are typable in JVML-0-C but not by the JVM. Then in Section 5.4 we analyze aspects of the design of the JVM verifier, as captured by JVML-0.

Section 6 presents some empirical data indicating that subroutines are rarely used in real Java code, and the kinds of situations that JVML-0 has trouble with are rarer still. However, they do occur and must be handled.

Section 4 presents the dynamic semantics of JVML-0-C, which are identical to those of JVML-0.

Section 7 introduces JVML-0-CB, a restriction of JVML-0-C to programs for which the stack size at each instruction is known.

In Section 8 we establish the properties identified as important by Stata and Abadi:

- *Type safety*. An instruction will never be given an operand stack with too few values on it, or with values of the wrong type.
- *Program counter safety*. Execution will not jump to undefined addresses.
- *Bounded operand stack*. The size of the operand stack will never grow beyond a static bound.

The first two properties hold for all JVML-0-C programs, but the third property holds only for JVML-0-CB programs. In JVML-0-C, with an unbounded operand stack, it is possible to write bytecode programs that use recursive subroutines in interesting ways, although such programs would never be generated by a standard Java compiler. We distinguish the two languages to demonstrate that the bound on the stack size is separable from the rest of the type system.

In Section 9 we show that a program's JVML-0 typing can be translated into a JVML-0-CB typing, implying that every JVML-0 program is also in JVML-0-CB.

3. STATIC SEMANTICS OF JVML-0-C

3.1 Preamble

We define substitutions S to map stack type variables to stack types and element type variables to element types in the natural way. The set of stack type variables and the set of element type variables are disjoint.

We define "$\Sigma_1 \leq \Sigma_2$" to mean instantiation, that is, "$\exists$ a substitution S. $S(\Sigma_1) = \Sigma_2$". The idea is that if a state can be given a type Σ_1, then it can also be given a type Σ_2 obtained from Σ_1 by instantiating the type variables. (Comparing this to STAL, where the scope of a type variable is a basic block of instructions, our approach is analogous to treating each instruction as a block of length one, with an explicit transfer from each instruction to its successor(s). This removes any need to explicitly identify or reason about blocks, which would require some overhead since JVML programs are simply a vector of instructions, and not naturally block-structured as in STAL.)

A program P is an array of instructions, i.e. a partial map from addresses to instructions. Addresses are isomorphic to integers,

but distinguished from them. We use + and the constant 1 for addresses as well as integers. ADDR is the set of all addresses.

3.2 Typing programs

A program P is well-typed if there exists a partial function Δ mapping instruction addresses to types, satisfying the judgement $\Delta \vdash P$, for which we have one rule:

$$\frac{\begin{array}{c}\forall x \in \text{dom VAR}.\ \varepsilon[x] = \text{TOP} \\ \Delta_1 \le (\text{NIL}, \varepsilon) \\ \forall i \in \text{dom } \Delta.\ \Delta, i \vdash P \\ \text{dom } \Delta \subseteq \text{dom } P\end{array}}{\Delta \vdash P}$$

The first and second hypotheses establish the initial conditions: the state-type for the first instruction must admit the type that has an empty stack and TOP types for all the local variables. (In other words, the local variables cannot be used in any meaningful way.) The third hypothesis ensures that types are checked at each program point using the rules below: $\Delta, i \vdash P$ means that the program type Δ is correct for P at instruction i. The fourth hypothesis ensures that addresses outside the program are not given types; thus, any instruction that could branch outside the program will not be typable, because each instruction's type rule has hypotheses which depend on the types of each of the successor instructions. We do not require every instruction to be typable; unreachable instructions need not have types. Of course, every constraint that mentions Δ_i implicitly requires i to be in the domain of Δ.

3.3 Widening

The widening operator $\prec$ is defined by the rules in Figure 4. It allows the type of any stack element or variable to be replaced with TOP. (Types inside a Cont do not change — allowing them to be widened would violate soundness, because the state accepted by the Cont might no longer be acceptable to the returned-to code.) This operator is used to "erase" some type information in the typing rule for `ret`, as discussed below.

$$\frac{\theta \prec \theta_w \quad T \prec T_w}{(\theta, T) \prec (\theta_w, T_w)}$$

$$\frac{\forall x \in \text{VAR}.\ T[x] \prec T_w[x]}{T \prec T_w}$$

$$\frac{\tau \prec \tau_w \quad \theta \prec \theta_w}{\tau \cdot \theta \prec \tau_w \cdot \theta_w}$$

$$\frac{}{\theta \prec \theta}$$

$$\frac{}{\tau \prec \tau}$$

$$\frac{}{\tau \prec \text{TOP}}$$

Figure 4: Widening rules

$$\frac{P[i] = \texttt{if}\ L \quad (\text{INT} \cdot \theta, T) = \Delta_i \quad \Delta_{i+1} \le (\theta, T) \quad \Delta_L \le (\theta, T)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{inc} \quad (\text{INT} \cdot \theta, T) = \Delta_i \quad \Delta_{i+1} \le (\text{INT} \cdot \theta, T)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{jsr}\ L \quad (\theta, T) = \Delta_i \quad \Delta_L \le (\text{Cont}(\Sigma) \cdot \theta, T) \quad \Delta_{i+1} \le \Sigma}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{ret}\ x \quad (\theta, T) = \Delta_i \quad T[x] = \text{Cont}(\theta_w, T_w) \quad (\theta, T) \prec (\theta_w, T_w)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{halt} \quad (\text{INT} \cdot \theta, T) = \Delta_i}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{pop} \quad (\tau \cdot \theta, T) = \Delta_i \quad \Delta_{i+1} \le (\theta, T)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{push0} \quad (\theta, T) = \Delta_i \quad \Delta_{i+1} \le (\text{INT} \cdot \theta, T)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{load}\ x \quad (\theta, T) = \Delta_i \quad \Delta_{i+1} \le (T[x] \cdot \theta, T)}{\Delta, i \vdash P}$$

$$\frac{P[i] = \texttt{store}\ x \quad (\tau \cdot \theta, T) = \Delta_i \quad \Delta_{i+1} \le (\theta, T[x \mapsto \tau])}{\Delta, i \vdash P}$$

Widening extends pointwise over state, stack and local variable types. Widening is reflexive. The last rule makes TOP an upper bound for element types.

Figure 3: JVML-0-C Type Rules

3.4 Typing instructions

For each kind of instruction, one of the rules in Figure 3 applies.

Each rule has a simple motivation: the type on entry to an instruction i must satisfy the immediate operational requirements of the instruction, and the types of i's successor(s) must be satisfied by the type of i's "after" state. The same reasoning even applies to `jsr`, where the "after" state contains a return address whose type encodes the requirements of the returned-to instruction. It also applies to `ret`, which introduces widening only to avoid a need for recursive types.

For example, the second hypothesis of the `inc` rule can be read "the state-type for instruction i implies that the top of the stack is an INT, that the rest of the stack has type θ, and that the local variables have types T". The third hypothesis of the `inc` rule can be read "the state-type for instruction i+1 admits a stack with an INT top element and tail of type θ, and local variables of types T".

The rule for `halt` ensures that an integer is left on top of the stack as the return value for the method.

The rule for `jsr` ensures that whenever the subroutine returns to i+1, it returns in state Σ, which will be acceptable at i+1.

The rule for `ret` ensures that the state on return is the state expected by the return address. The third hypothesis ensures that the type of variable x is a return address, and the fourth

$$\frac{P[pc] = \texttt{inc}}{P \vdash \langle pc, f, n \cdot s\rangle \rightarrow \langle pc + 1, f, (n + 1) \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{pop}}{P \vdash \langle pc, f, v \cdot s\rangle \rightarrow \langle pc + 1, f, s\rangle}$$

$$\frac{P[pc] = \texttt{push0}}{P \vdash \langle pc, f, s\rangle \rightarrow \langle pc + 1, f, 0 \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{load}\ x}{P \vdash \langle pc, f, s\rangle \rightarrow \langle pc + 1, f, f[x] \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{store}\ x}{P \vdash \langle pc, f, v \cdot s\rangle \rightarrow \langle pc + 1, f[x \mapsto v], s\rangle}$$

$$\frac{P[pc] = \texttt{if}\ L}{P \vdash \langle pc, f, 0 \cdot s\rangle \rightarrow \langle pc + 1, f, s\rangle}$$

$$\frac{\begin{array}{c}P[pc] = \texttt{if}\ L \\ v \neq 0\end{array}}{P \vdash \langle pc, f, v \cdot s\rangle \rightarrow \langle L, f, s\rangle}$$

$$\frac{P[pc] = \texttt{jsr}\ L}{P \vdash \langle pc, f, s\rangle \rightarrow \langle L, f, (pc + 1) \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{ret}\ x}{P \vdash \langle pc, f, s\rangle \rightarrow \langle f[x], f, s\rangle}$$

Figure 5: Dynamic Semantics for JVML-0/JVML-0-C

hypothesis ensures that the current state type matches the type expected on return.

Instead of these two hypotheses, it would be more natural to simply write "$T[x] = \text{Cont}(\theta, T)$". However, this would require $T[x]$ to be a recursive type. Introducing recursive types would complicate the type system, and the extra programs that could be typed by such a system may not be practically interesting. Therefore, we avoid recursive types by allowing type information to be erased using the widening operator. This allows us to type $T_w[x]$ as TOP, effectively preventing the returned-to code from having access to its own address in x. Other variables or stack locations can also be erased, as in the JVM and JVML-0.

STAL solves the problem in a similar way; the register holding the return address is never given a type by the code that is returned to. In fact, giving a value type TOP is always analogous to simply omitting it from the STAL type signature.

Note that allowing generalization in any of the hypotheses for `ret` (e.g. "$(\theta, T) \leq (\theta_w, T_w)$") leads to unsoundness. Intuitively, constant code can be polymorphic, and therefore instantiated at different types by different predecessors, but a return address value is not polymorphic, because the type variables in the type of the returned-to code have already been bound in the type environment of the caller.

4. DYNAMIC SEMANTICS OF JVML-0-C

The rules presented in this section are identical to those of JVML-0.

Execution starts at address 1; if $1 \notin \text{dom}\ P$, then no transitions can occur and the program cannot typecheck. The initial state is $\langle 1, f, \epsilon\rangle$ for some arbitrary f, where ϵ is the empty stack.

The small-step transition rules are given in Figure 5.

5. COMPARING JVML-0-C WITH JVML-0 AND THE JVM

5.1 Typing subroutines with "abnormal" behavior

Bytecode subroutines are used by Java compilers to compile try-finally statements. The statement "try S finally F" executes S and then F, no matter how control leaves S; if S throws an exception or returns from the method, F will still be executed "on the way out". Typically the code for F is placed in a subroutine, and called on each of the exit paths for S.

i	$P[i]$	Δ_i
0	push0	(NIL, [0: α_0, 1: α_1, 2: α_2])
1	store 0	(INT · NIL, [0: α_0, 1: α_1, 2: α_2])
2	push0	(NIL, [0: INT, 1: α_1, 2: α_2])
3	store 1	(INT · NIL, [0: INT, 1: α_1, 2: α_2])
4	load 1	(NIL, [0: INT, 1: INT, 2: α_2])
5	inc	(INT · NIL, [0: INT, 1: α_1, 2: α_2])
6	store 1	(INT · NIL, [0: INT, 1: α_1, 2: α_2])
7	load 0	(NIL, [0: INT, 1: INT, 2: α_2])
8	if 12	(INT · NIL, [0: INT, 1: INT, 2: α_2])
9	jsr 14	(NIL, [0: INT, 1: INT, 2: α_2])
10	load 1	(NIL, [0: α_0, 1: INT, 2: α_2])
11	halt	(INT · NIL, [0: α_0, 1: α_1, 2: α_2])
12	jsr 14	(NIL, [0: INT, 1: INT, 2: α_2])
13	goto 4[2]	(NIL, [0: INT, 1: INT, 2: α_2])
14	store 2	(Cont(NIL, [0: INT, 1: INT, 2: TOP]) · NIL, [0: INT, 1: INT, 2: α_2])
15	load 0	(NIL, [0: INT, 1: INT, 2: Cont(NIL, [0: INT, 1: INT, 2: TOP])])
16	if 4	(INT · NIL, [0: INT, 1: INT, 2: Cont(NIL, [0: INT, 1: INT, 2: TOP])])
17	ret 2	(NIL, [0: INT, 1: INT, 2: Cont(NIL, [0: INT, 1: INT, 2: TOP])])

Figure 7: A bytecode program with an "abnormal" subroutine

The question arises, "what happens if control does not leave F 'normally'"? Consider the code in Figure 6. This program would loop forever, but one could replace "`done`" and "`!done`" with more complicated boolean conditions yielding various results. Sun's Java compiler (supplied with JDK 1.1) produces bytecode similar to the JVML-0-C code in Figure 7 (shown with one possible JVML-0-C typing). Notice how the return address type in local variable 2 is widened to TOP on return so that the return address type need not be recursive. The local variable 0 corresponds to `done`, and local variable 1 corresponds to `x`.

The 'finally' subroutine at 14 may not return; instead, it exits by simply jumping out! A similar "exit without returning" can occur when a subroutine throws an exception that is caught in the method. These abnormal exits can make a subroutine appear to be invoked recursively, or a subroutine can appear to return to code that was not its immediate caller. JVML-0 cannot type such programs, because it relies on control leaving a subroutine only through a `ret` instruction. In JVML-0-C, because we do not maintain a notion of the "current subroutine", there is no need to know when the subroutine is exited. The return address value is simply discarded if it will not be used.

5.2 Typing control-flow merges

In the JVM, if there is a control flow merge where two predecessors have incompatible types for a particular stack element, the program is rejected. However, merging of two incompatible types for a local variable yields TOP. If the value of the local variable is dead at the merge point, then the program can pass verification.

In JVML-0-C, this is achieved by typing a local variable with an arbitrary type variable if the local variable is dead at the instruction. This type variable can be instantiated in a different way by each predecessor of the instruction; i.e. incompatible types can be merged if the variable or stack element is dead at the merge point.

5.3 Beyond the JVM

JVML-0-C is more flexible than the JVM in some ways. A bytecode verifier based on JVML-0-C would allow constructs that could be useful to advanced compilers for Java or other languages.

JVML-0-C allows a subroutine to have multiple entry and/or exit points (JVML-0 requires a single entry point, and the JVM requires a single entry and a single exit). In some situations this

```
boolean done = false;
int x = 0;
while (true) {
    try {
        x++;
        if (done) return x;
    } finally {
        if (!done) continue;
    }
}
```

Figure 6: A Java program with an "abnormal" subroutine

[2] Unfortunately JVML-0 is so simple that it cannot faithfully encode "goto" (an unconditional branch that does not depend on the type of the following instruction). We take a little license by using it here; the type rule for "goto" would simply require $\Delta_{target} \leq \Delta_{source}$.

i	$P[i]$	Δ_i	
0	push0	(NIL, [0: α_0, 1: α_1])	x = 0;
1	store 0	(INT · NIL, [0: α_0, 1: α_1])	
2	jsr 5	(NIL, [0: INT, 1: α_1])	call subroutine;
3	load 0	(NIL, [0: INT, 1: α_1])	return x;
4	halt	(INT · NIL, [0: α_0, 1: α_1])	
5	load 0	(Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	subroutine: {
6	inc	(INT · Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	x = x + 1;
7	store 0	(INT · Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	
8	load 0	(Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	if (x == 0) {
9	if 11	(INT · Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	
10	jsr 5	(Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	call subroutine;
11	store 1	(Cont(ϕ, [0: INT, 1: TOP]) · ϕ, [0: INT, 1: α_1])	}
12	ret 1	(ϕ, [0: INT, 1: Cont(ϕ, [0: INT, 1: TOP])])	}

Figure 8: A recursive subroutine

would reduce duplication of code.

Our type system allows more polymorphism in subroutines than the JVM-like systems. In those systems a subroutine is polymorphic in the types of the local variables it does not touch. In our type system, polymorphic values (including values on the operand stack) can be manipulated by a subroutine, as long as the operations do not constrain the value's type. For example, the program in Figure 2 contains a subroutine that swaps the values of two stack elements regardless of their type.

JVML-0-C allows subroutines to be truly recursive. If a static bound on the stack size is required for implementation reasons, i.e., programs are restricted to JVML-0-CB, then recursive subroutines are not very useful. If such a bound is not required, then potentially useful recursive subroutines can be typed: see Figure 8. This program calls the subroutine at 5 recursively, incrementing local variable 0 each time, until the value reaches zero. (Of course this never actually happens, but the type system is oblivious to that fact.[3])

5.4 Assessing subroutine constraints in the JVM verifier

Because the JVM-like systems try to build an explicit model of the subroutine call stack, they impose a LIFO discipline on subroutine calls and returns. This necessitates subroutines being named and associated with instructions — the purpose of the "labeling rules" in JVML-0. These rules can reject programs if no consistent labeling can be found (examples are programs that jump out of subroutines as mentioned above). JVML-0 also uses "strong labeling" rules to construct an abstraction of the call stack at each program point. These rules reject all programs with recursive subroutine calls. JVML-0-C shows that explicit modeling of a subroutine call stack, and all the associated machinery, is not necessary for a sound type system.

In order to maintain the LIFO behavior of subroutines, the JVM-like systems must also carefully restrict the use of return address values. In JVML-0, the type of a return address value is of the form `ret-from` L. An instruction "`jsr` L" pushes a value of type `ret-from` L into the stack, and a "`ret`" instruction requires its argument to be of type `ret-from` L' where L' is the first instruction of the current subroutine. In contrast, in JVML-0-C the return address type simply specifies the state expected by the returned-to code, and is not bound to any particular subroutine. Because the JVML-0 type does not encode the type of the return state, an extra nonlocal constraint in their type rule for `ret` is used to ensure that the state on return is acceptable to the returned-to code. The constraint is "nonlocal" in the sense that it quantifies over all instructions in the program. JVML-0-C does not require such nonlocal constraints.

Without additional constraints in JVML-0, there remains the possibility that a subroutine L could return to an address generated by a call to L that was not the most recently executed call to L. Therefore their rule for `jsr` L prohibits any stack element or local variable from being given the type `ret-from` L on entry to L — ensuring that only the most recently generated return address is available for use.[4] In JVML-0-C nothing prevents a subroutine from returning to an "old" address, but it is still guaranteed to be safe. In other words, LIFO subroutine behavior is not necessary for a sound type system.

JVML-0 and the JVM have explicit constraints preventing a return address value from being used by a `ret` instruction more than once. Interestingly, this property is also enforced indirectly by JVML-0-C: it is a direct consequence of the absence of recursive types in JVML-0-C. We could probably remove this restriction by adding recursive types, but there is no justification for the extra complexity.

[3] Unfortunately JVML-0 cannot encode subtraction or a comparison with nonzero.

[4] In a similar way, the constraints in Freund and Mitchell's type system for object initialization [FM98] ensure that only one object of a particular type is available for initialization.

6. EMPIRICAL DATA ON THE USE OF SUBROUTINES

We examined all the code in the standard JDK 1.1.4 `classes.zip` library. There are about 2.2 MB of bytecode instructions, corresponding to 10986 method bodies. Of these, 175 contain at least one subroutine. Seven of these contain code that would not typecheck with naïve labeling rules based on those of JVML-0. All these examples consist of subroutines inside exception catch blocks: if an exception is thrown inside the subroutine, control passes to the exception handling block and the subroutine is deemed to have "returned", although no `ret` instruction has been executed.

In retrospect, it seems that adding subroutines to the bytecode language may have been a poor design choice. In the 1-2% of methods that use them, the compiler could have generated explicit flags recording the caller and used conditional branches to simulate a return. The need for polymorphism could be avoided by duplicating more code and/or using more local variables. It seems likely that the extra code size and runtime costs would have been minimal overall, and bytecode verification would have been greatly simplified. Of course, the need to support legacy code means that we are stuck with the language as it stands.

7. BOUNDING THE STACK SIZE IN JVML-0-CB

The JVM requires that a static bound on the size of the stack be known, and that the height of the stack be known at each program point. This may simplify language implementations.

7.1 Additional type rule

We can enforce this constraint by using the following rule for $\Delta \vdash P$:

$$\frac{\begin{array}{c}\forall x \in \mathrm{dom}\ \mathrm{VAR}.\ \varepsilon(x) = \mathrm{TOP} \\ \Delta_1 \le (\mathrm{NIL}, \varepsilon) \\ \forall i \in \mathrm{dom}\ \Delta.\ \Delta, i \vdash P \\ \mathrm{dom}\ \Delta \subseteq \mathrm{dom}\ P \\ \forall i \in \mathrm{dom}\ \Delta.\ \exists n, \tau_1, \ldots, \tau_n, \mathrm{T}_i.\ \Delta_i = (\tau_1 \cdot \ldots \cdot \tau_n \cdot \mathrm{NIL}, \mathrm{T}_i)\end{array}}{\Delta \vdash P}$$

The extra fifth hypothesis forces each stack type to be a finite string of element types.

We call the language of programs that satisfy this rule JVML-0-CB. Clearly JVML-0-C is a superset of JVML-0-CB.

8. SOUNDNESS

The main result is practically identical to that of Stata and Abadi, and establishes the type safety and program counter safety properties. Their theorem statement mentions a type judgement relating a dynamic stack to a stack type. To avoid introducing such judgements for our theorem statement, we assume that the method is supposed to return INT. (In this language, the only alternative is to not return anything.)

Theorem 1 (Soundness). Given $\Delta \vdash P$, execution of P only stops if a `halt` instruction is reached, and when it stops there is an integer on top of the stack.

$\forall pc, f_0, f, s.$
$(P \vdash \langle 1, f_0, \epsilon \rangle \rightarrow^* \langle pc, f, s \rangle$
$\wedge\ \nexists pc', f', s'.\ P \vdash \langle pc, f, s \rangle \rightarrow \langle pc', f', s' \rangle)$
$\Rightarrow P[pc] = \mathtt{halt} \wedge \exists n, s''.\ s = n \cdot s''$

For the proof, we define an extended dynamic semantics for typed programs that carries around some type information at runtime. We show that the soundness result holds for this semantics, and then show that standard dynamic semantics corresponds to the extended semantics with the type information erased.

The proof machinery also enables a demonstration that, in the extended dynamic semantics, a given return address value can be used by a `ret` at most once. Because we show that the extended dynamic semantics bisimulates the standard dynamic semantics, the result extends immediately to the standard dynamic semantics.

9. TRANSLATING JVML-0 TYPES

JVML-0-CB is a superset of JVML-0. We show this by translating a program's JVML-0 type assignment into a valid JVML-0-CB type assignment for the same program. The details of this translation are given in Appendix B.

The correctness of the translation depends on the fact that JVML-0 subroutines cannot use "nonlocal returns", which suggests that extending JVML-0 to handle nonlocal returns would be nontrivial. The key problem with JVML-0 is that the $C_{P,i}$ abstraction of the call stack at instruction i does not encode enough information to determine, statically, whether a nonlocal return should be allowed. For example, if $C_{P,i} = \mathrm{L} \cdot \mathrm{L}_1 \cdot \mathrm{L}_2$, then we know that subroutine L is active, but we do not know if L_1 is active. (This abstraction could be generated when L_1 and L_2 each call L.) Therefore a nonlocal return at i using a value of type `ret-from` L_1 may or may not be valid.

10. IMPLEMENTATION ISSUES

The complete verification process must first generate the type information, and then check it. Only the type checking phase needs to be trusted.

10.1 Type checking

Checking types is straightforward, given the type assignment Δ and for each `jsr` instruction, the "return to" type Σ used in its type rule. With these in hand, all terms occurring in the type rules are known. The equality and widening constraints are easily checked by structural induction. The instantiation constraints are checked by trying to unify the two terms while treating the variables of the right hand side as constants.

We have implemented the typechecker in C, totaling about 350 lines of code, and used it to check the examples in this paper.

10.2 Type inference

The type system is closely related to type systems incorporating polymorphic recursion [3]. Full type inference for polymorphic recursion has been shown to be undecidable in the general case [3]. In practice one would obtain types by applying some less general algorithm, such as an algorithm analogous to the JVM's own verifier, and translate the types into JVML-0-C to be checked for safety.

11. RELATED WORK

The type system is very similar to that of the Stack-based Typed Assembly Language of Morrisett et al. [6]. They both rely on polymorphic recursion to handle loops, and they both type code addresses with the type of the state that the code expects on entry. STAL's existential types for closures are unnecessary since JVML does not support closures. (Java does support "inner classes", a limited kind of closures, but they are translated to JVML objects.)

Freund and Mitchell [2] investigated adding object initialization constraints to JVML-0. It does not appear to be difficult to adapt their approach to JVML-0-C.

Qian has formulated type checking and inference for a very rich subset of JVML [7]. However, his machinery to handle subroutines that do not exit cleanly is rather complex [Section 10.6 of [7]]. For example, for each instruction it constructs a mapping from pairs of subroutine addresses to sets of local variables.

Hagiya and Towaza [4] have proposed another method for checking bytecode subroutines. Although it can handle some kinds of nonlocal returns, it relies on constructing the set of all possible "call stacks" at each instruction, and therefore cannot type programs with an infinite number of possible subroutine call stacks (such as the valid Java program in Section 5.1).

12. CONCLUSIONS

We have developed a type checker for bytecode subroutines by applying familiar concepts long known to the programming language community — continuations and polymorphic recursion — rather than creating ad-hoc structures based on the JVM's intuitive approach to approximating run-time behavior. This enables a direct comparison between the two general approaches, and we believe that the results show this "constraint based" type checking is fundamentally simpler and more powerful.

We see no reason why this approach should be any more difficult than other approaches to extend to full JVML. In particular, since the set of JVML object types and the subtype relation between them is known *a priori* by the bytecode verifier, we would not expect particular difficulties integrating a treatment of objects into JVML-0-C.

Our work has several practical implications. In general it seems that the design of "low level" typed languages such as bytecode would benefit from experiences with the type systems of "high level" languages. For JVML itself, our approach might lead to a design for a simple checking engine whose implementation would be easier to trust than that obtained by other approaches. Furthermore, if such an engine were widely used, then the extra programs it admits could be useful. For example, an optimizing compiler could better compact bytecodes by using subroutines with multiple entries and exits, or by collapsing similar code sequences into a polymorphic subroutine and making use of the extra polymorphism we provide. Compilers for other languages that target Java bytecode might also be able to exploit the extra flexibility.

13. ACKNOWLEDGEMENTS

I am supported by a Microsoft graduate fellowship. I would like to thank the CMU POP group for constantly bombarding me with types, and especially Robert Harper for his help and advice.

14. REFERENCES

[1] D. Dean, E. Felten and D. Wallach. Java Security: From HotJava to Netscape and beyond. 1996 IEEE Symposium on Security and Privacy, May 1996.

[2] S. Freund and J. Mitchell. A type system for object initialization in the Java bytecode language. Proceedings of the ACM SIGPLAN '98 Conference on Object-Oriented Programming Systems, Languages and Applications, October 1998.

[3] F. Henglein. Type inference with polymorphic recursion. TOPLAS, Volume 15, No. 2, 1993.

[4] M. Hagiya and A. Towaza. On a new method for dataflow analysis of Java Virtual Machine subroutines. Proceedings of the Fifth International Static Analysis Symposium, Springer-Verlag LNCS, 1998.

[5] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley.

[6] G. Morrisett, K. Crary, N. Glew and D. Walker. Stack-Based Typed Assembly Language. 1998 Workshop on Types in Compilation.

[7] Z. Qian. A formal specification of Java virtual machine instructions. Formal Syntax and Semantics of Java, Springer-Verlag LNCS, 1998.

[8] R. Stata and M. Abadi. A type system for Java bytecode subroutines. Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1998.