Towards Array Bound Check Elimination in Java™ Virtual Machine Language*

Hongwei Xi and Songtao Xia
Oregon Graduate Institute
{hongwei,sxia}@cse.ogi.edu

Abstract

The Java Virtual Machine Language (JVML) is a platform independent programming language at bytecode level. A Java program is compiled into JVML bytecode in a standard Java implementation, which may be transferred across networks to various hosts and then interpreted by the Java Virtual Machine. Note that these hosts may not trust the source of the bytecode. For safety concerns, JVML disallows out-of-bounds array access. In practice, this is enforced through run-time array bounds checking. Unfortunately, this practice can be prohibitively expensive in cases involving numerical computation such as animation.

In this paper, we propose a type-theoretic approach to eliminating run-time array bound checks in JVML and demonstrate that the property of safe array access can be readily captured with a restricted form of dependent type system and therefore enforced through type-checking. We focus on a language JVMLa, which is basically a subset of JVML with array access instructions, and prove that the execution of a well-typed JVMLa program can never cause memory violations.

1 Introduction

Java is an object-oriented programming language designed to support multiple host architectures and allow secure delivery of software components [13]. One key step to achieving this objective is the introduction of Java Virtual Machine (JVM) [14]. In the standard implementation of Java, a Java program is compiled into bytecode in Java Virtual Machine Language (JVML), which can then be interpreted by JVM.

JVML bytecode is platform independent and can be transferred across networks to a host which may not trust the source of the bytecode. This makes it necessary to check for various potential safety violations in JVML bytecode. A significant concern in this context is memory violations, including both run-time type errors and out-of-bounds array accesses. Clearly, we can insert run-time checks to prevent memory violations from happening, but this practice can significantly slow down bytecode execution. The introduction of Java virtual machine verifier can partially address the issue. The key point is that type safety of JVML bytecode can be practically checked before execution. JVML bytecode that can pass JVM verifier is guaranteed to cause no type errors at run-time. Unfortunately, potential out-of-bounds array accesses cannot be detected through JVM verifier and run-time array bound checks are needed.

Array bounds checking can be expensive in practice. For instance, array bounds checking in an implementation of the matrix multiplication algorithm can take up to 50% of the execution time. In general, tasks involving numerical computation such as animation can suffer a great deal from run-time array bounds checking. This seems to be a significant reason for not disallowing out-of-bounds array access in many unsafe programming languages such as C and C++. As a consequence, it is common to encounter inscrutable crashes of C or C++ programs in practice.

A traditional approach to eliminating run-time array bound checks is based on flow analysis [3, 4, 5]. Although this is a fully automatic approach, it is also greatly limited and sensitive to program structures. For instance, it seems unlikely that such an approach can be adopted in
practice to eliminate array bounds checking in an implementation of binary search on arrays. In addition, an approach based on flow analysis often provides no or little feedback to the programmer indicating why a particular array bound check in a program cannot be eliminated. Therefore, this approach offers little help for detecting program errors caused by potential out-of-bounds array accesses.

High Performance Compiler for Java (HPCJ) compiles JVM bytecode into native code before executing the bytecode, eliminating array bound checks through flow analysis on the bytecode. This is a radically different approach to implementing JVM in which is usually done through interpretation. Also it seems significantly more difficult to eliminate array bound checks in JVM bytecode than in Java source programs since some structures in Java programs are destroyed when they are compiled into bytecode. In general, a compiler like HPCJ sacrifices binary portability but provides more sophisticated optimizations. Clearly, a limited platform coverage and increased maintenance costs have to be traded against the performance improvements.

A type-based approach is introduced in [15] for eliminating array bound checks. The key point behind this approach is the use of a restricted form of dependent types to capture the property of safe array access and then enforce it through type-checking. We briefly explain this through the following code in de Caml, a dialect of DML [16].

```
let icopy src dst =
  for i = 0 to vect_length(src) - 1 do
    dst..(i) <- src..(i)
  done

withtype {n:nat} 'a vect(n) -> 'a vect(n) -> unit

;;
```

The icopy function copies an integer array into another one. We use the double dot `..` notation for safe array access. The withtype clause is a type annotation and the type `n:nat `a vect(n) -> `a vect(n) -> unit states that for every natural number n, the function icopy takes two vectors of length n and returns a value of type unit. It can be readily observed that both array accesses `src..(i) and dst..(i) are safe since i is always within the bounds of src and dst. This property is captured by the type system of de Caml, and therefore no run-time array bounds checking is needed after the code passes type-checking in de Caml. Notice that it is impossible to determine whether dst..(i) is safe if the type annotation is not supplied.

In this paper, we apply this idea to JVMLa, a subset of JVML, presenting an approach towards eliminating array bound checks in JVML. Inspired by [7, 10], we design a type system for JVMLa which is expressive enough to capture the memory safety property of JVMLa bytecode, where memory safety consists of both type safety and safe array access. We are thus able to enforce memory safety of bytecode through type-checking and eliminate run-time array bound checks. This approach is particularly suitable for mobile code since the site which executes the code may not trust the source of the code. With such a type system, the verification of the memory safety of mobile code can be performed through type-checking, regardless the origin of the code. We now present a short JVMLa program before going into further details. Notice that the simplicity of this example is solely for the sake of illustration purpose and should not be interpreted as the limitation of our approach.

The JVMLa program in Figure 1 roughly corresponds to the above program in de Caml. Note that a double slash `//` starts a comment line. The formal representation of this program in JVMLa will be given in Section 4, and we now informally explain what the program means. For every integer i, the type `int(i)` is a singleton type in which the only element is integer i. Also we use type `intarray(n)` for integer arrays of size n. Note that v0, v1, ..., v4 are local variables. Each label in the code is associated with a dependent type. For instance, the dependent type associated with the label icopy means that the top two elements of the current stack are of type `intarray(n)` for some natural number n. We use \(\Phi\) for the rest of stack of which we have no knowledge. Also the dependent type associated with the label `loop` basically means that there exist natural numbers n and k satisfying \(k \leq n\) such that v0, v1, ..., v4 are of types `intarray(n)`, `intarray(n)`, `int(n)` and `int(k)`, respectively. Clearly, the type of v0 depends on the value of the integer in v2. The type system of JVMLa guarantees that this property is satisfied when the code execution reaches the label `loop`. The JVMLa code is well-typed. As a consequence (which we will explain later), this implies that the index is within the bounds of the array when either the instruction on line 14 or line 19 is executed. In
00. icopy: \{n:nat\} \{sp: intarray(n) :: intarray(n) :: @\}
01. astore 0 // v0 <- src
02. astore 1 // v1 <- dst
03.aload 0
04. arraylength
05. istore 2 // 03, 04, 05: store the array length into v2
06. push 0
07. istore 3 // 06, 07: initialize the loop count: v3 <- 0
08. loop: \{n:nat, k:nat | k <= n\}
   \{v0: intarray(n), v1: intarray(n), v2: int(n), v3: int(k)\}
09. iload 3 // load array index onto the stack
10. iload 2 // load array size onto the stack
11. isub // calculate the difference
12. ifeq finish // difference equals zero: done
13. aload 0 // push the src array pointer onto the stack
14. iload 3 // push the index i onto the stack
15. iload // push the content in src[i]
16. istore 4 // store the content into v4
17. aload 1 // push the dst array pointer onto the stack
18. iload 3 // push the index i onto the stack
19. iload 4 // push the content onto the stack
20. iastore // store the content into dst[i]
21. push 1
22. iload 3
23. iadd
24. istore 3 // 20, 21, 22, 23: increment loop count by 1
25. goto loop // loop again
26. finish: []
27. halt // program terminates

Figure 1: A copy function implemented in JVMLa

In other words, it is unnecessary to perform run-time array bound checks when these instructions are executed. We have thus eliminated array bounds checking in this program.

The main contribution of the paper is the design of the type system of JVMLa and the formalism which leads to the soundness proof of the type system. We feel that we have made an important step towards eliminating array bounds checking in Java virtual machine, which is a difficult question of significant practical relevance.

The organization of the paper is as follows. In Section 2, We introduce the language JVMLa and present both of its dynamic and static semantics. The type-checking in JVMLa involves constraint satisfaction, which is then explained in Section 3. The type system of JVMLa is involved, and we present an example in Section 4 to explain in details how type-checking in JVMLa is performed. The soundness of the type system of JVMLa is proven in Section 5, and an extension to handling \textit{jump-to-subroutine} feature in JVM is given in Section 6. We then mention in Section 7 and 8 some issues on implementing JVMLa and generating JVMLa-like bytecode. The rest of the paper deals with related work and concludes.

2 The language JVMLa

We find it instructive to concentrate on the essence of the approach to eliminating array bound checks that we will introduce shortly. We form a lan-
Figure 2: The syntax for the index language

```
index objects \( x, y \) ::= a | i | x + y | x - y | x \ast y | x \div y 
index propositions \( P \) ::= x < y \mid x \leq y \mid x \geq y \mid x > y \mid x = y \mid x \neq y \mid 
\top \mid \bot \mid P_1 \land P_2 \mid P_1 \lor P_2 
index sorts \( \gamma \) ::= \text{int} \mid \{ a : \gamma \mid P \} 
index contexts \( \phi \) ::= \cdot \mid \phi, a : \gamma \mid \phi, P 
```

Figure 3: JVMLa syntax

guage JVMLa for this purpose, which is basically a minimal subset of JVML with some array access instructions.

For the sake of simplicity, only integer arrays are allowed in JVMLa. Notice that the object-oriented features in JVML have little to do with array bounds checking and are thus excluded. We do not handle function calls, either. Nonetheless, we point out that the type system, like that of DML, is to guarantee arguments of correct types are supplied when a function call is made. For instance, when the `icopy` function in Figure 1 is called, it must be provable in the type system that the top two elements on the current stack are two integer arrays of the same size; if this cannot be proven, a run-time check is needed to verify this. Also we state that many details omitted in this paper for the sake of space limitation can be found in [19].

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we may form a type `intarray(x)` such that every array of this type is an integer array of size \( x \), which \( x \) is the expression on which this type depends. We use the name `type index object` for such an expression. Also we can refine the type `int` into infinitely many singleton types `int(x)`, where \( x \) ranges over all integers. The value of every expression of type `int(x)` equals \( x \). There are various compelling reasons for imposing restrictions on expressions which can be chosen as type index objects. A novelty in DML [16] is to require that type index objects be drawn only from a given constraint domain. This is crucial to obtaining a practical type-checking algorithm. For our purpose, we restrict type index objects to integers.

The syntax for type index objects is presented in Figure 2, where we use `a` for type index variables, and `i` for fixed integers. Note that the language for type index objects is typed. We use `sorts` for the types in this language in order to avoid potential confusion. We use `·` for the empty index context and omit the standard sorting rules for this language. We also use certain transparent abbreviations, such as \( 0 \leq x < y \) which stands for \( 0 \leq x \land x < y \). The subset sort \( \{ a : \gamma \mid P \} \) stands for the sort for those elements of sort \( \gamma \) satisfying the proposition \( P \). For example, we use `nat` as an abbreviation for \( \{ a : \text{int} \mid a \geq 0 \} \).

The syntax of JVMLa is given in Figure 3 and the meaning of JVMLa instructions is informally
aload n push the array pointer in variable n onto the stack
astore n pop an array pointer from the stack and store it into variable n
iload n push the integer in variable n onto the stack
istore n pop an integer from the stack and store it into variable n
iaload pop an index i and an array pointer a from the stack and store the content in the ith cell of a onto the stack
iastore pop an integer, an index i and an array pointer a from the stack and store the integer into the ith cell of a
iadd pop two integers i and j from the stack and store i + j onto the stack
isub pop two integers i and j from the stack and store i − j onto the stack
imul pop two integers i and j from the stack and store i * j onto the stack
idiv pop two integers i and j from the stack and store i/j onto the stack
goto L jump to L
ifeq L pop an integer from the stack and branch to L if zero
ifne L pop an integer from the stack and branch to L if nonzero
pop pop away the top element from the stack
push i push integer i onto the stack
newarray pop an integer i from the stack and allocate an integer array of size i on the heap and push the array pointer onto the stack
arraylength pop an array pointer from the stack and push the size of the array onto the stack
halt terminate execution

Figure 4: A summary for instructions

presented in Figure 4. A program is a pair (Λ, I), where I is a sequence of general instructions and Λ associates every label in I with a type τ. Note that a label is treated as a general instruction (its operational semantics is simply to increment the program count by 1). In the external language, we write a type following each label to mean that the label is associated with the type following it. Also we use length(I) for n if I is of form \( Lins_0; \ldots; Lins_{n−1} \), and I(i) for \( Lins_i \) if 0 \( \leq i < \) length(I), where \( Lins_i \). We write \( I^{-1}(L) \) for \( \text{if} \) i if some \( Lins_i \) is L. This is well-defined since we require that all labels in a program be distinct.

The following erasure function \( || \cdot || \) erases all syntax related to type index objects, transforming types into type erasures.

\[
||\text{unit}|| = \text{unit} \quad ||\text{int}(x)|| = \text{int} \quad ||\sigma|| = \text{unit} \\
||\text{intarray}(x)|| = \text{intarray} \quad ||\exists a : \gamma.\tau|| = ||\tau||
\]

In the following presentation, we also write \( \text{int} \) for \( \exists a : \text{int}.\text{int}(a) \) if \( \text{int} \) is treated as a type.

JVMLa is to be interpreted with a stack-based abstract machine. The execution of a JVMLa program is associated with a frame, which consists of a set of local variables and a stack. We assume there is a fixed number \( n_e \) of local variables. We do not consider stack overflow in this paper and therefore assume the existence of a stack of unlimited depth.

2.1 Dynamic semantics

We use an abstract state machine for assigning operational semantics to JVMLa. This is a standard approach. A machine state \( M \) is a triple (\( \mathcal{H}, \mathcal{V}, \mathcal{S} \)), where \( \mathcal{H} \) and \( \mathcal{V} \) are finite mappings standing for heap and the set of local variables, respectively, and \( \mathcal{S} \) is a list standing for stack. The domain \( \text{dom}(\mathcal{H}) \) of \( \mathcal{H} \) is a set of heap addresses and the domain \( \text{dom}(\mathcal{V}) \) is \( \{0, 1, \ldots, n_e − 1\} \), where \( n_e \) is the number of local variables. We do not specify how a heap address is represented, but the reader can simply assume it to be a natural number. We use \( h \) for a heap address, \( i \) for an integer and \( hi \) for a value which is either a heap address or an integer. Given \( h \in \text{dom}(\mathcal{H}) \), \( \mathcal{H}(h) \) is a tuple \((i_0, \ldots, i_{n-1})\) such that every \( i_k \) is an integer for \( k = 0, \ldots, n − 1 \). Given \( k \in \text{dom}(\mathcal{V}) \), \( \mathcal{V}(k) \) is either a heap address or an integer. Also we use \( h_i :: \ldots :: h_i \) for a stack where the top \( n \)
\[
P(pc) = \text{iaload} \quad \mathcal{H}(h) = (i_0, \ldots, i_{n-1}) \quad 0 \leq j < n \\
(p, (\mathcal{H}, \mathcal{V}, j :: h = s)) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, i_j :: s)) \quad \text{(eval-iaload)}
\]

\[
P(pc) = \text{istore} \quad \mathcal{H}(h) = (i_0, \ldots, i_{n-1}) \quad 0 \leq j < n \\
\mathcal{H}' = \mathcal{H}[h \mapsto (i_0, \ldots, i_{j-1}, i, i_{j+1}, \ldots, i_{n-1})] \\
(p, (\mathcal{H}, \mathcal{V}, i :: j :: h :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}', \mathcal{V}, \mathcal{S})) \\
P(pc) = \text{iadd} \\
(p, (\mathcal{H}, \mathcal{V}, i :: j :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, (i + j) :: \mathcal{S})) \quad \text{(eval-iadd)}
\]

\[
P(pc) = \text{idiv} \\
(p, (\mathcal{H}, \mathcal{V}, i :: j :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, (i/j) :: \mathcal{S})) \quad j \neq 0 \quad \text{(eval-idiv)}
\]

\[
P(pc) = \text{ifeq} L \\
(p, (\mathcal{H}, \mathcal{V}, i :: j :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \\
(p, (\mathcal{H}, \mathcal{V}, 0 :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \quad \text{(eval-ifeq-t)}
\]

\[
P(pc) = \text{ifeq} L \\
(p, (\mathcal{H}, \mathcal{V}, i :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \\
(p, (\mathcal{H}, \mathcal{V}, i :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \quad i \neq 0 \quad \text{(eval-ifeq-f)}
\]

\[
P(pc) = \text{goto} L \\
(p, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \\
(p, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \quad \text{(eval-goto)}
\]

\[
P(pc) = \text{newarray} \\
(p, (\mathcal{H}, \mathcal{V}, i :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \\
(p, (\mathcal{H}, \mathcal{V}, i :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, \mathcal{S})) \quad i \geq 0 \quad h \text{ is new} \quad \text{(eval-newarray)}
\]

\[
P(pc) = \text{arraylength} \\
(p, (\mathcal{H}, \mathcal{V}, h :: \mathcal{S})) \rightarrow_P (p + 1, (\mathcal{H}, \mathcal{V}, n :: \mathcal{S})) \\
P(pc) = \text{halt} \quad \langle p, \mathcal{M} \rangle \rightarrow_P \text{HALT} \quad \text{(eval-halt)}
\]

Figure 5: Evaluation rules for JVMLa

Elements are \(hi_1, \ldots, hi_n\) and the rest of the stack is represented by \(\mathcal{S}\).

Given a program \(P = (A, I)\), a P-snapshot \(Q\) is either HALT or a pair \((pc, \mathcal{M})\) such that \(0 \leq pc < \text{length}(I)\). The relation \((pc, \mathcal{M}) \rightarrow (pc', \mathcal{M}')\) means that the current machine state \(\mathcal{M}'\) transforms into \(\mathcal{M}'\) after executing the instruction \(I(pc)\) and the instruction count is set to \(pc'\). We present some typical evaluation rules for JVMLa in Figure 5. We do not consider garbage collection in this abstract machine, and therefore the heap can only be affected by the memory allocation instruction \textbf{newarray}.

Given a finite mapping \(f\) and an element \(x\) in the domain of \(f\), we use \(f(x)\) for the value to which \(f\) maps \(x\), and \(f[x \mapsto v]\) for the mapping such that

\[
f[x \mapsto v](y) = \begin{cases} 
  f(y) & \text{if } y \neq x; \\
  v & \text{if } y = x.
\end{cases}
\]

Clearly, \(f[x \mapsto v]\) is also meaningful when \(x\) is not already in the domain of \(f\). In this case, we simply extend the domain of \(f\) with \(x\).

We present some explanation on some of the evaluation rules. The rule \textbf{eval-iaiostore} is for evaluating the instruction \textbf{istore}. This rule is applicable only if the current stack is of form \(i :: j :: h :: \mathcal{S}\), that is, the top three elements are an integer, an integer and a heap address, and \(\mathcal{H}\) maps the heap address to an integer array \((i_0, \ldots, i_{n-1})\) of size \(n\), and \(0 \leq j < n\) holds. If the required conditions do not hold, the execution stalls. We will form a type system for JVMLa so that the execution of well-typed JVMLa programs never stalls. Since out-of-bounds array access stalls execution, this implies that no out-of-bounds array access can occur during the execution of \textit{well-typed} JVMLa programs.

The rule \textbf{eval-newarray} is for evaluating the instruction \textbf{newarray}. A fresh new heap address \(h\) is added into the domain of \(\mathcal{H}\) after the instruction
is executed, and \( h \) is then mapped to an integer array initialized with zeros.

The interpretation of the rest of evaluation rules should now be clear to the reader. The complete list of evaluation rules can be found in [19].

2.2 Static semantics

We present some typical typing rules for JVMLa in this section. Constraints on integers are generated during type-checking in JVMLa. We postpone the treatment of constraint satisfaction until Section 3 in order for a gentle presentation. However, we informally explain the need for constraints through the JVMLa code in Figure 1. Notice that the third local variable \( v3 \) is assumed to be of type \( \text{int}(k_1) \) for some natural number \( k_1 \) when the execution reaches the label \( \text{loop} \). The type of \( v3 \) changes into \( \text{int}(k_1 + 1) \) after the execution of the instruction on line 23. Then the execution jumps back to the label \( \text{loop} \). It must be verified (among many other requirements) that \( v3 \) is of type \( \text{int}(k_2) \) for some natural number \( k_2 \).

Therefore, we must prove that \( k_1 + 1 \) is a natural number under the condition that \( k_1 \) is a natural number. This is a constraint, though it is trivial in this case. In general, type-checking in JVMLa involves solving a great number of constraints of this form.

We have so far introduced various forms of judgments, some of which have yet to be defined later. In Figure 7, we summarize the meaning of these judgments informally.

We present some explanation on some typing rules. Generally speaking, an index variable context \( \phi \) contains some assumption on index variables declared in \( \phi \). For instance, \( n : \text{nat}, k : \text{nat}, k \leq n \) is an index variable context which states that \( n \) and \( k \) are of sort \( \text{nat} \) and \( k \leq n \) holds, that is, \( n \) and \( k \) are natural number satisfying \( k \leq n \).

The rule (type-istore) states that in order to type an instruction sequence of form iastore; \( I \), it is required that the current frame be of form \((V, \tau :: \text{int}(x) :: \text{intarray}(n) :: S)\) such that \( I \) is typable under \( \phi; (V, S) \) and the erasure of \( \tau \) is \( \text{int} \) and \( 0 \leq x < n \) can be proven under \( \phi \). This means that we must be able to verify that the value of index is within the array bounds when indexing an array. Obviously, a natural question at this moment is what to do if we cannot prove the index is within the bounds of the array. We give some explanation on the rule (type-ifeq) before answering this question.

In order to type ifeq \( L; I \), it is required that the stack type is of form \( \text{int}(x) :: S \) for some index \( x \). We first find the state type \( \text{state}(\exists \bar{a} : \bar{\gamma}; F) \) associated with \( L \) in the label map \( \Lambda \), and then verify that there exist \( \bar{x} \) such that \( F[\bar{a} \mapsto \bar{x}] \) (the result from substituting \( \bar{x} = x_1, \ldots, x_n \) for \( \bar{a} = a_1, \ldots, a_n \) in \( F \)) is satisfied under \( \phi; x = 0; (V, S) \). It is important to notice that \( x = 0 \) is in the index variable context since ifeq \( L \) branches only if \( x \) is zero. Also we need to verify that \( I \) is typable under context \( \phi; x \neq 0; (V, S) \), where \( x \neq 0 \) is assumed since \( I \) is executed only if ifeq \( L \) does not branch.

The typing rules for other conditional branching instructions should be straightforward. Suppose we cannot prove under context \( \phi \) that an index of type \( \text{int}(x) \) is less than the size of an array of type \( \text{intarray}(n) \). We compute the array length and store the difference between the index and the array length on top of that stack, which is of type \( \text{int}(i - n) \). We then do ifgte \( L \), that is, jump to \( L \) if greater than or equal to zero, where \( L \) is the label of an entry to some code handling array bounds violation. This allows us to assume that \( i - n < 0 \) when type-checking the rest of code. In other words, we can readily prove that the index is less than the array size after inserting a runtime check. The important point is that we do not have to insert such checks if we have some other ways to prove that the index is within the array bounds in the type system, which is often the case in practice.

The rule (type-goto) for typing goto \( L \) involves a judgment of form \( \text{goto} \vdash_A I \), for which we have the following typing rules.

\[
\text{goto} \vdash_A \text{halt} \quad \text{goto} \vdash_A I \\
\Lambda(L) = \text{state}(\exists \bar{a} : \bar{\gamma}; F) \quad \bar{a} : \bar{\gamma}; F \vdash_A I \\
\text{goto} \vdash_A \bar{L}; I
\]

This basically means that instructions following a goto are not accessible until a label appears.

The rest of the typing rule can be interpreted in a similar manner. The complete list of typing rules can be found in [19].

We use the judgment \( \vdash (\Lambda, I) \text{[well-typed]} \) to mean that the program \( P = (\Lambda, I) \) is well-typed. The following rule is for such a judgment, where
\[
\begin{align*}
\phi, a : \gamma; (V[n \mapsto \tau], S) & \vdash I \quad \text{(type-open-var)} \\
\phi; (V[n \mapsto \exists a : \gamma. \tau], S) & \vdash I \\
\phi; a : \gamma; (V, \tau :: S) & \vdash I \\
\phi; (V, \exists a : \gamma.x :: S) & \vdash I \\
\phi; (V, \text{int} :: S) & \vdash \lambda I \quad \phi \models 0 \leq x < n \\
\phi; (V, \text{int}(x) :: \text{intarray}(n) :: S) & \vdash \lambda \text{iaload}; I \\
||r|| & = \text{int} \quad \phi \models 0 \leq x < n \quad \phi; (V, S) \vdash \lambda I \\
\phi; (V, \text{int}(x) :: \text{int}(y) :: S) & \vdash \lambda \text{idiv}; I \\
\phi; (V, \text{int}(x/y) :: S) & \vdash \lambda I \quad \phi \models y \neq 0 \\
\phi; (V, \text{int}(x) :: \text{int}(y) :: S) & \vdash \lambda \text{idiv}; I \\
\text{goto} & \vdash \lambda I \quad \Lambda(L) = \text{state}(\exists a : \tau.F) \quad \phi \vdash \exists : \tau' \quad \phi; (V, S) \vdash F[\bar{a} \mapsto \bar{x}] \\
\phi; (V, S) & \vdash \lambda \text{goto} \; L; I \\
\Lambda(L) & = \text{state}(\exists a : \tau.F) \quad \phi \vdash \exists : \tau \\
\phi, x = 0; (V, S) & \vdash F[\bar{a} \mapsto \bar{x}] \\
\phi, x \neq 0; (V, S) & \vdash I \\
\phi; (V, \text{int}(x) :: S) & \vdash \lambda \text{if}eq \; L; I \\
\phi; (V, \text{int}(n) :: S) & \vdash \lambda I \\
\phi; (V, \text{intarray}(n) :: S) & \vdash \lambda \text{arraylength}; I \\
\phi \models n \geq 0 \quad \phi; (V, \text{intarray}(n) :: S) & \vdash \lambda I \\
\phi; (V, \text{int}(n) :: S) & \vdash \lambda \text{newarray}; I \\
\Lambda(L) & = \text{state}(\exists a : \tau'.F') \quad \phi \vdash \exists : \tau' \\
\phi; F & \vdash \lambda L; I \\
\text{goto} & \vdash \lambda I \\
\phi; F & \vdash \lambda \text{halt}; I \\
\vdash (\Lambda, I)[\text{well-typed}] & \quad \text{The program } P = (\Lambda, I) \text{ is well-typed.}
\end{align*}
\]

Figure 6: Some typing rules for JVMLa

Judgment form          Judgement meaning
\[\phi \models P\]  The proposition \(P\) holds under the context \(\phi\) in the integer domain.
\[\phi \models \tau_1 \leq \tau_2\]  The type \(\tau_1\) coerces into the type \(\tau_2\) under the context \(\phi\) modulo constraint satisfaction.
\[\phi; V \models V'\]  The type of \(V(i)\) under the context \(\phi\) coerces into \(V'(i)\) for every \(0 \leq i < n_v\) modulo constraint satisfaction.
\[\phi; S \models S'\]  The stack type \(S\) coerces into the stack type \(S'\) under the context \(\phi\).
\[\phi; (V, S) \models (V', S')\]  This means both \(\phi; V \models V'\) and \(\phi; S \models S'\) are derivable.
\[\phi; F \vdash \lambda I\]  The instruction sequence \(I\) is typable with respect to the label map \(\Lambda\).
\[\text{goto} \vdash \lambda I\]  The instruction sequence \(I\), which is after a \text{goto}, is typable.
\[\vdash (\Lambda, I)[\text{well-typed}]\]  The program \(P = (\Lambda, I)\) is well-typed.

Figure 7: A summary for various forms of judgments

8
$V_{\text{empty}}$ maps $n$ to $\text{unit}$ for $n = 0, \ldots, n_v - 1$.

\[
\vdash (V_{\text{empty}}, \bullet) \vdash I
\]

\[
\vdash (\Lambda, I)[\text{well-typed}]
\]

3 Type coercion

We present the rules for type coercion in this section, which clearly exhibit the involvement of constraint satisfaction in type-checking JVMLa programs. In the presence of dependent types, it is no longer trivial to determine whether two types are equivalent. For instance, it is necessary to prove $1 + 1 = 2$ in order to claim $\text{int}(1 + 1)$ is equivalent to $\text{int}(2)$. Similarly, type coercion also involves constraint satisfaction. The need for type coercion is mainly for typing conditional and unconditional jumps as shown in the rules (type-ifeq), (type-ifne) and (type-ifnull) (the derivation of a judgment of form $\phi; F \vdash F'$ involves type coercion).

We present the syntax for constraints as follows, where $P$ stands for index propositions.

index constraints $\Phi ::= P \mid P \supset \Phi \mid \forall a : \gamma. \Phi$

We use $\phi \models \Phi$ to mean that the constraint $\Phi$ is satisfied under the index context $\phi$, that is, the formula $\phi(\Phi)$ is satisfiable in the domain of integers, where $\phi(\Phi)$ is defined below.

\[
\phi(\Phi) = \Phi
\]

\[
\phi(\phi, \{a : \gamma \mid P\}) = (\phi)(\forall a : \text{int}. \Phi)
\]

\[
\phi(\phi, \{a : \gamma \mid P\}) = (\phi)(P \supset \Phi)
\]

For instance, the satisfiability relation $a : \text{nat}, b : \text{int}, a + 1 = b \models b \geq 0$ holds since the following formula is true in the integer domain.

\[
\forall a : \text{int}. a \geq 0 \supset \forall b : \text{int}. a + 1 = b \supset b \geq 0
\]

The rule (coerce-state) deserves some explanation. Informally speaking, a state type $\text{state}(\exists a : \gamma. F)$ is “stronger” if $\exists a : \gamma. F$ is “weaker”. The intuition is that a state type roughly represents the notion of code continuation.

We define substitutions for index variables as follows.

index substitutions $\theta ::= [] \mid [\theta[a \mapsto x]]$

We omit the details on how substitution is performed, which is standard. Given an expression $e$ such as a type or a state, we use $e[\theta]$ for the result from applying $\theta$ to $e$.

4 An example

It is simply too overwhelming to formally demonstrate how type-checking is performed in JVMLa because of the involvedness of the type system. Instead, we present some key ingredients in type-checking the JVMLa program in Figure 1.

Let $(\Lambda, I)$ represent the program, that is, we have the following, where $\text{Lins}_i$ is the instruction on line $i$ in the program for $i = 0, \ldots, 27$, and $V_{\text{loop}}$ is the mapping \{ $v_0 : \text{intarray}(n), v_1 : \text{intarray}(n), v_2 : \text{int}(n), v_3 : \text{int}(k)$ \} as explained below. For $i = 0, \ldots, 27$, we write $I_i$ for the suffix of $I$ starting at $\text{Lins}_i$, where $I = \text{Lins}_0, \text{Lins}_1, \ldots, \text{Lins}_{27}$.

In the derivation $D$ of $\text{goto} \vdash I$, there are subderivations $D_i$ with conclusions of form $\phi_i; (V_i, S_i) \vdash I_i$ for $i = 0, \ldots, 27$. Some of these $\phi_i, V_i, S_i$ are listed in Figure 9. Recall that $V_{\text{empty}}$ maps $n$ to $\text{unit}$ for $n = 0, \ldots, n_v - 1$. We write $\{ v_0 : \tau_1, \ldots, v_{n_v} : \tau_k \}$ for the $V$ such that $V(n) = \tau_i$ if $n = \tau_i$ for some $1 \leq i \leq k$ and $V(n) = \text{unit}$ otherwise. For instance, the derivation $D_{10}$ is of the following form, where the last applied rule is (type-iaload).

\[
D_{10} \quad \|\text{int}(k)\| = \text{int} \quad \phi_{15} \models 0 \leq k < n
\]

\[
\phi_{15}; (V_{15}, S_{15}) \vdash \text{iaload}; I_{10}
\]

Note $\phi_{15} = n : \text{nat}, k : \text{nat}, k \leq n, k \neq n$, and therefore, the constraint $\phi_{15} \models 0 \leq k < n$ holds in the integer domain. This implies that the index is within array bounds when iaload is executed.

The derivation $D_{25}$ is also interesting, which is of the following form. The applied rule is (type-goto).

\[
\phi_{25} \models n : \text{nat} \quad \phi_{25} \models k + 1 : \{ a : \text{nat} \mid a \leq n \}
\]

\[
\phi_{25}; (V_{25}, S_{25}) \models \{ V_{\text{loop}}[n, k \mapsto n, k + 1], \bullet \}
\]

\[
\text{goto} \vdash \text{finish}; \text{halt}
\]

The constraint $\phi_{25} \models k + 1 : \{ a : \text{nat} \mid a \leq n \}$ breaks into $\phi_{25} \models k + 1 : \text{nat}$ and $\phi_{25} \models k + 1 \leq n$. The former is obviously satisfied. For the latter, please notice that $k \leq n$ and $k \neq n$ are assumed in $\phi_{25}$ and thus $k < n$ holds, which implies $k + 1 \leq n$.

5 Soundness

We establish the soundness of the type system of JVMLa in this section. Let us recall that a machine state $M$ is a triple $(H, V, S)$, where $H$ and
<table>
<thead>
<tr>
<th>\phi \vdash \tau : * \quad \phi \models x = y</th>
<th>(\text{coerce-unit})</th>
<th>(\text{coerce-int})</th>
</tr>
</thead>
<tbody>
<tr>
<td>\phi \models x = y \quad \phi \models \text{int}(x) \leq \text{int}(y)</td>
<td>(\text{coerce-array})</td>
<td></td>
</tr>
<tr>
<td>\phi \models \text{intarray}(x) \leq \text{intarray}(y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\phi, a : \gamma \models \tau_1 \leq \tau_2 \quad \phi \models \exists a : \gamma. \tau_1 \leq \tau_2</td>
<td>(\text{coerce-exi-ivar-1})</td>
<td></td>
</tr>
<tr>
<td>\phi \vdash x : \gamma \quad \phi \models \tau_1 \leq \tau_2[a \rightarrow x] \quad \phi \models \tau_1 \leq \exists a : \gamma. \tau_2</td>
<td>(\text{coerce-exi-ivar-r})</td>
<td></td>
</tr>
<tr>
<td>\phi, a_2 : \tau_2 \vdash \xi : \tau_1 \quad \phi, a_2 : \tau_2, F_2 \models F_1[a_2 \rightarrow \xi]</td>
<td>(\text{coerce-state})</td>
<td></td>
</tr>
<tr>
<td>\phi \models \text{state}(\exists a_1 : \tau_1. F_1) \leq \text{state}(\exists a_2 : \tau_2. F_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\phi; V \models V' \quad \phi; S \models S'</td>
<td>(\text{coerce-frame})</td>
<td></td>
</tr>
<tr>
<td>\phi; (V, S) \models (V', S')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\phi \models V(i) \leq V'(i) \text{ for } 0 \leq i &lt; n \quad \phi; V \models V'</td>
<td>(\text{coerce-vars})</td>
<td></td>
</tr>
<tr>
<td>\phi; S \models \bullet</td>
<td>(\text{coerce-stack-1})</td>
<td></td>
</tr>
<tr>
<td>\phi \models \tau \leq \tau' \quad \phi; S \models S'</td>
<td>(\text{coerce-stack-2})</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Type coercion rules for JVMLa

\( V \) are finite mappings for the heap and local variables, and \( S \) is a list for the stack.

Given a machine state \( M = (H, V, S) \) and a frame \( F = (V, S) \), the judgment \( H \models (V, S) : (V, S) \) means that \( M \) models the frame \( F \). The rules for deriving such a judgment is presented in Figure 10. Also we write \( M \models \phi ; F \) if \( \vdash \theta : \phi \) is derivable for some substitution \( \theta \) and \( M \) models \( F[\theta] \).

Given a derivation \( D \) of \( \vdash P[\text{well-typed}] \) for program \( P = (\Lambda, I) \), we use \( D_i \) for the greatest subderivation of \( D \) with conclusion \( \phi_i ; F_i \vdash I_i \) for some \( \phi_i ; F_i \), where \( I_1 \) is the suffix of \( I \) starting at the ith instruction. Also we write \( D(i) \) for \( \phi_i ; F_i \), that is, \( D(i) \vdash I_i \) is the conclusion of \( D_i \).

**Lemma 5.1 (Main lemma)** Assume that \( D \) is a derivation of \( \vdash P[\text{well-typed}] \) for program \( P = (\Lambda, I) \) and \( M \models D(pc) \) is derivable for some \( pc \). Then \( (pc, M) \rightarrow \text{HALT} \) or \( (pc, M) \rightarrow p \) (\( pc', M' \)) for some \( M' \) and \( pc' \) such that we can construct a derivation for \( M' \models D(pc') \).

**Proof** The lemma follows from a structural induction on the derivation \( D \).

**Theorem 5.2 (Progress)** Let \( M_0 \) be a machine state and \( P = (\Lambda, I) \) be a program such that \( \vdash P[\text{well-typed}] \) is derivable. If \( (0, M_0) \rightarrow p_\gamma (pc, M) \) then either \( (pc, M) \rightarrow \text{HALT} \), or \( (pc, M) \rightarrow_p (pc', M') \) for some \( pc' \) and \( M' \). In other words, the execution of a well-typed program in JVMLa either halts normally, or runs forever.

**Proof** The theorem immediately follows from an application of Lemma 5.1.

It is clear from the evaluation rules in Figure 5 that out-of-bounds array access is disallowed in JVMLa. Therefore, Theorem 5.2 guarantees that the execution of a well-type JVMLa program cannot lead to run-time memory violation. This implies that memory safety of JVMLa programs can be enforced through static type-checking.

### 6 Extension

The instructions \( jsr \) and \( ret \) in JVML complicate a JVML verifier implementation significantly as stated in [10]. This, however, is not the case for JVMLa, which can readily handle \( jsr \) and \( ret \). The execution of \( jsr L \) pushes the program counter of the
<table>
<thead>
<tr>
<th>No.</th>
<th>( \phi )</th>
<th>( V )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(k) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(n) :: \text{int}(k) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(n - k) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{intarray}(n) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(k) :: \text{intarray}(n) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int} :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k) )</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{intarray}(n) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(k) :: \text{intarray}(n) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int} :: \text{int}(k) :: \text{intarray}(n) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(1) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(k) :: \text{int}(1) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \text{int}(k + 1) :: \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k), v_4 : \text{int} )</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>( n : \text{nat}, k : \text{nat}, v_0 : \text{intarray}(n), v_1 : \text{intarray}(n) )</td>
<td>( \bullet )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( k \leq n, k \neq n )</td>
<td>( v_2 : \text{int}(n), v_3 : \text{int}(k + 1), v_4 : \text{int} )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Contexts \( \phi_n; (V_n, S_n) \) for \( n = 8, 8', 9, \ldots, 25 \)

\[
I = \text{icopy; ins}_1; \ldots; \text{ins}_7; \text{loop; ins}_9; \ldots; \text{ins}_{25}; \text{finish; halt}
\]

\[
\Lambda(\text{icopy}) = \text{state}(\exists n : \text{nat}.(V_{\text{empty}}. \text{intarray}(n) :: \text{intarray}(n) :: \bullet))
\]

\[
\Lambda(\text{loop}) = \text{state}(\exists n : \text{nat}. \exists k : \{a : \text{nat} \mid a \leq n\}.(V_{\text{loop}}. \text{intarray}(n) :: \text{intarray}(n) :: \bullet))
\]

\[
\Lambda(\text{finish}) = \text{state}(V_{\text{empty}}, \bullet))
\]
\[ \begin{align*}
\mathcal{H} &= i : \text{int}(i) \quad \text{(heap-int)} \quad \mathcal{H}(h) = (i_0, \ldots, i_{n-1}) \\
\mathcal{H} &= h : \text{int-array}(n) \quad \text{(heap-intarray)} \\
\vdash x : \gamma & \quad \mathcal{H} \models hi : \tau[a \mapsto x] \\
\mathcal{H} &\models \exists a : \gamma . \tau \\
\mathcal{H} &\models V(i) : V(i) \text{ for } 0 \leq i < n_v \\
\mathcal{H} &\models V : V \\
\mathcal{H} &\models S : \bullet \quad \mathcal{H} &\models hi : \tau \\
\mathcal{H} &\models S : \tau :: S \\
\mathcal{H} &\models V : V \\
\mathcal{H} &\models S : S \\
\mathcal{H} &\models (V, S) : (V, S) \quad \text{(heap-stack-1)} \quad \text{(heap-stack-2)} \quad \text{(heap-frame)}
\end{align*} \]

Figure 10: Rules for modeling states

\[ \begin{align*}
\Lambda(L_1) &= \text{state}(\exists a_1 : \gamma_1, F_1) \\
\Lambda(L_2) &= \text{state}(\exists a_2 : \gamma_2, F_2) \\
\phi; (V, \Lambda(L_2) :: S) &\models F_1[a_1 \mapsto x] \\
\phi; (V, \Lambda(L_2) :: S) &\models F_2[a_2 \mapsto \text{ins}; I] \\
\phi; (V, \Lambda(L_2) :: S) &\models F[a \mapsto x] \\
\phi; (V, S) &\models \text{got o} + I \quad \text{(type-jsr)}
\end{align*} \]

\[ \begin{align*}
\phi; (V, S) &\models \text{jr;} L_1; L_2; \text{ins}; I \\
\phi; (V, S) &\models \text{ret n;} I \quad \text{(type-ret)}
\end{align*} \]

Figure 11: Typing rules for jr and ret

next instruction onto the stack and then jumps to label \( L_i \), and the execution of ret \( n \) jumps to the label stored the \( n \)th local variable. We omit the straightforward evaluations rules for these two instructions. The typing rules for these instructions are presented in Figure 11. Notice that we assume that the instruction following a \( \text{jr} \) is always associated with a label. Note \( \Lambda(L_2) :: S \) in the rule \( \text{(type-jsr)} \) reflects the fact that a label is pushed onto the stack after a \( \text{jr} \) instruction is executed.

7 Implementation

There is certain amount of nondeterminism in the the typing rules of JVMLa. In the implementation, we impose some restriction to eliminating the nondeterminism. For instance, when both of the rules \( \text{coerce-exi-ivar-l} \) and \( \text{coerce-exi-ivar-r} \) are applicable, we choose the former over the latter. For each of the rules \( \text{type-open-var} \) and \( \text{type-open-stack} \), we apply it whenever it is applicable. More details can be found in [19]. We have finished a prototype implementation for JVMLa and verified many examples [18]. Several examples are basically taken from the bytecode generated using a compiler released by Sun. It seems unlikely, as indicated by these realistic examples, that an approach based on flow analysis can automatically eliminate array bound checks in JVML bytecode highly effectively. This yields a strong support for studying the type-based approach proposed in this paper. The current implementation, written in Objective Caml, corresponds to the formal development of JVMLa tightly. We are now trying to add more features of JVM into this implementation.

8 Type annotation

An immediate question regarding the applicability of the proposed approach is how to generate the state types attached to the labels in a program. One possibility is to synthesize the information with some flow analysis. This, however, seems to a very limited approach. We are currently exploring an alternative, requiring the programmer to provide some annotations in programs for eliminating array bounds checking. The is
{n:nat}
int bsearch(key: int, vec: <int> array(n)) {

  /* init: is a key word indicating variable declaration */
  init: int low, mid, high, x;;

  /* arraylength computes the length of an array */
  low = 0; high = arraylength(vec) - 1;

  /* invariant: indicates a program invariant */
  invariant:
  [i:int, j:int | 0 <= i <= n, 0 <= j+1 <= n] (low: int(i), high: int(j))
  while (low <= high) {
    mid = (low + high) / 2; x = vec[mid];
    if (key == x) { return mid; }
    else if (key < x) { high = mid - 1; }
    else { low = mid + 1; }
  }
  return -1;
}

Figure 12: An implementation of binary search in Xanadu

an effective approach as is argued in [15]. With a dependently typed language at bytecode level, the provided annotations can then be compiled into bytecode. Following the DML example [16], we have designed a dependently typed language Xanadu, which has a Java-like syntax. The code in Figure 12 is an implementation of binary search on an integer array. The state type following invariant: is a loop invariant which states that the values of low and high must equal i and j, respectively, for some integers i and j such that both 0 \leq i \leq n and 0 \leq j + 1 \leq n hold, where n is the size of the searched vector. We have prototyped a compiler for Xanadu, which translates such invariants into state types at bytecode level. In the binary search example, the invariant suffices to prove that the array access vec[mid] is always safe, that is, the value of mid is always within the bounds of the array. The important point is that the invariant is carried to the bytecode level so that the memory safety of the generated bytecode can be verified through type-checking at a site which may not trust the source of the code. In other words, it is not enough to simply eliminate array bound checks in a source program; we must provide evidence in the generated code that such checks are safely eliminated so that the correctness of the elimination can be verified independently.

Though it seems burdensome to write annotations, the programmer is not forced to do so. In the binary search example, the program can still run without the invariant annotation but some run-time checking is then needed to enforce the safety of the array access. However, we feel that it is a good practice to write annotations since they can also serve as informative program documentation facilitating program understanding and evolution. We point out that this example also demonstrates some limitation of automatic flow analysis since it seems highly unlikely that the invariant in this example can be effectively synthesized in practice. Such examples (e.g., quicksort, FFT, etc.) are abundant in practice. In general, it is heuristic at best to synthesize loop invariants with flow analysis, and this often makes it difficult to predict whether a particular array access can be eliminated.

9 Related work

A typed assembly language (TAL) is first introduced in [7], and a stack-based typed assembly language (STAL) is then introduced in [6]. A type system similar to STAL is developed in [10]
for JVML-0-C, which is basically a minimal subset of JVML capturing the feature of *jump-to-subroutine*. The type systems of these languages suffice to guarantee type safety of programs. Unfortunately, the property of safe array access cannot be captured in these type systems. As a consequence, it is impossible to type optimized programs in which some array bound checks are eliminated. This is a serious limitation.

A restricted form of dependent type system is developed in [13] for eliminating array bounds checking in a functional programming language. This notion of types is carried further into an assembly language in [17], leading to a dependently typed assembly language (DTAL), where the type system is adequate for capturing memory safety of code. In this paper, we apply the approach developed in [17] to a bytecode language JVMLa, which is basically a subset of JVML with array access instructions. We believe that the type system of JVMLa is the first for a JVML-like language that can capture memory safety property of programs.

The notion of proof-carrying code (PCC) [9] is also suitable for eliminating array bound checks. This is demonstrated by Touchstone, a compiler for the Safe C programming language [8]. Given TAL is already adequate for this purpose, we feel that Touchstone’s approach to handling type safety is too refined. Also we are not clear at this moment on how to use Touchstone’s approach for dealing with the *jump-to-subroutine* feature in JVML. On the other hand, if one prefers, one may generate proofs for certifying the well-typedness of JVMLa programs. This opens an alternative to Touchstone’s method for generating proof-carrying code.

As mentioned in the introduction, JVML verifier [14] currently cannot detect potential out-of-bounds array accesses and every array access in JVM is wrapped with array bound checks to enforce memory safety. This makes it *impossible* to eliminate array bound checks in a standard implementation of JVM since an array access instruction with bound checks is treated as an atomic instruction in this setting. Instead, we propose to separate array access instructions from array bound checks and use a type system to enforce memory safety. It is this separation that allows us to eliminate array bound checks in the case where we can prove in the type system that they are redundant.

Recently, there has been a great deal of work on forming type systems for JVML-like languages, such as [11, 12, 2] in addition to JVML-0-C. A key objective is to formalize the specification for JVM verifier in contrast to its current prose description in English. In addition to advocating this, this paper also intends to strengthen the JVM verifier so that more safety checks can be performed statically. This can not only produce more robust code but also enhance the efficiency of code since many run-time checks are saved.

We are currently extending JVMLa to full JVML. Like JVML-0-C, we see no reasons why this approach is more difficult for extension than other approaches. In particular, we feel that this approach is largely orthogonal to the treatment of objects in Java, and thus expect the integration of objects into JVMLa to be smooth.

There are many potential applications for the type system of JVMLa. For instance, we may use the type system to encode some information obtained from flow analysis and then verify the information through type-checking. A particular interesting question at this moment is whether the type system can encode the results generated by the algorithm in [5] for eliminating array bound checks. Also we may use some extension of JVMLa as the target language for compiling programs with annotations [16, 1], translating source-level annotations into the dependent types in JVMLa programs. This is especially helpful in the case where it is difficult to eliminate array bound checks automatically.

Java is becoming a popular general-purpose programming language, but currently it seems significantly slower than C++, especially, in the field of numerical computation. This makes it a highly relevant research topic to study array bounds checking elimination in JVM if Java is to penetrate into this field. A similar problem along this line is to eliminate pointer casts in JVML, which seems very likely to be handled through the use of an adequate type system for this purpose.

**Acknowledgments**

We gratefully acknowledge our discussion with Jim Hook and Andrew Tolmach regarding the subject of the paper. Also we thank PacSoft center at Oregon Graduate Institute for the support we received, and especially Jim Hook for his encouragement.
References


