Index types for JVM method invocation

Songtao Xia
sxia@cse.ogi.edu

James Hook
hook@cse.ogi.edu

June 1, 2000

Abstract

Language based security is a promising technique that supports secure and efficient applications on the internet. Certain security information in the compiled mobile code is downloaded along with the code and is verified before the code is run by the downloader. Annotating code with index types is one particular form of language-based security; it allows strong program properties to be asserted and checked. We report our experience of applying a restricted form of dependent types to JVMLi, a subset of Java Virtual Machine Language (JVML). The result is a type system for JVMLi that can guarantee safe elimination of some of the array- bound checks. We show how our restriction on the type terms results in a clean coding of method invocation in this first-order language with explicit control. We prove the soundness of our system and that a well-typed program is guaranteed to be array-access safe.

1 Introduction

With such words as "explosive rise of the Internet" in their introductory paragraphs, many recent research papers from the programming language community target the security-performance tradeoff problems in the context of mobile computing. The techniques described in these papers are generally categorized as language-based security approaches [7].

In its most common form, a language-based security approach extracts extra information from a program written in a high-level language and packages this information in the object code compiled from the program. This extra information, called a certificate, is supplied with the code to allow independent validation of critical program properties, such as security. Several forms of certificates have been investigated, including proofs of properties, type annotations and annotation with data structures arising from program analysis. When the code is downloaded to a different site, the certificate is downloaded along with the code. At the receiver's site, a verifier will inspect the code and the certificate to predict whether the code will compromise the security of the host system. When the receiver's site has ensured that the program indeed satisfies the properties specified by the certificate, and that these properties guarantee a certain level of security, it may then accordingly relax some expensive security checks in order to gain performance. Examples of language-based security approaches are Necula's proof-carrying code (PCC) [11], Morrisett's typed assembly languages (TAL) [9, 10], and Kozen's efficient code certification (ECC) [6].

Traditionally program properties have been calculated and exploited by tools designed to work together in an environment of mutual trust. With mobile code the analysis and exploitation of properties is done by separate tools and without trust. This raises many questions about what program properties may be efficiently calculated, compactly expressed, and efficiently verified. These questions hinge on the logical language used to encode program properties and the algorithmic properties of that language. Intuitively, the stronger the assertions a logic can state, the more the receiver can rely on static checks that are only performed once rather than dynamic checks that are performed repeatedly to establish the desired security policies. On the other hand, powerful program properties tend to be expensive or even impossible to check. In the examples we mentioned above, PCC uses first-order logic to code proofs, which is rich in expressive power. TAL annotates the object code with type information and type-checks the assembly code. ECC keeps simple type information regarding the structure and intention of a program in the object code, which facilitates efficient verification.

In this paper, the certificates are expressed as indexed type annotations on Java virtual machine code. It is expected that these annotations will be produced by compilers based on standard program analysis techniques, possibly augmented by high-level programming language type annotations. The type framework used for the abstract machine instructions is influenced strongly by Morrisett's typed assembly language and by Xi's and Zenger's work [20, 23, 22, 26, 25] on indexed types in functional languages.

In their doctoral theses [20, 26], Xi and Zenger independently demonstrated how index types could be used to optimize code, among other applications. The index type technique combines traditional, Hindley-Milner style type checking algorithms with the generation of a constraint set, and the use of an external constraint solving system to determine if the constraints are satisfiable. As a consequence, the tractability of the index type analysis depends on the tractability of the constraint solving problems that are generated by the type checking algorithm. If the constraint domain is undecidable, such as arbitrary arithmetic expressions, then type checking is potentially incomplete. If the constraint domain is restricted, say to Presberger arithmetic, a complete algorithm may exist, but tractability can only be established through experience (that is, it is known that the worst case complexity of these algorithms is extreme, but in practice their performance is highly dependent on the intrinsic difficulty of the specific constraint sets generated). Preliminary results, based on Xi's experience with the de CAML implementation of index types, suggests that indexed type checking is practical for an interesting set of programs, including many of Okasaki's functional data structures [14, 21].

This paper is focusing on properties that allow the elimination of array bounds checks in a JVM-like language. These properties are validated by an extension of traditional type checking with linear arithmetic constraint solving. We expect, but do not demonstrate in this paper, that these properties can be generated by a compiler using conventional analysis techniques. Xi has demonstrated in other contexts that high-level langauge annotations can be introduced to allow programmers to express these properties directly in the source language [20].

The indexed type approach presented here is a refinement of the TAL approach. Compared with TAL we expect to slightly increase the cost of certificate generation and validation, while significantly increasing the expressive power of the certificates. We believe, but have not formally demonstrated, that this approach is more expressive than ECC and that the certificates are more easily generated than those for PCC.

Types provide a mechanism to statically characterize the range of values that may occur dynamically. Index types refine a type system by allowing it to statically make finer distinctions about the dynamic behavior of the program. In a general index typed setting, any type can be refined by an index set known to the constraint solver. In this paper, only two types will be indexed, integer and integer array. Both types will be refined by indexing them with integers.

The refined integer type yields singleton types containing exactly the index value. That is, the type containing the infinite set of integers is refined into an infinite set of types, each containing exactly one integer. Intuitively, this will make it difficult to statically give an expression an index type (since we have to evaluate it!), but it does assure that if two expressions have the same index type then they have the same value. The difficulty in knowing values statically in the index type scheme is overcome by characterizing the indexes of values symbolically and recording the known relationships between them in a constraint set. The values of the program variables that will only be known dynamically can be characterized symbolically at type checking time (or code verification time). It is not necessary to know exactly what indexes x and y have to know that their sum has index $x + y$. If the type checker can deduce from the constraint set that a value has an index that is within the bounds of an array, the type checker can guarantee the safety of that array access.

Integer arrays are refined by indexes indicating their dimension. Other refinements are possible, but they are not considered here.

This paper presents JVMLi, our second step in the integration of index type checking with certification of JVM code. In an earlier system [24], JVMLa, we focused only on annotations that could be established without inter-procedural information. In this paper we extend that work to include index types characterizing properties of the return values of methods. For example, the system in this paper can characterize that the result of the factorial function is always strictly positive. We report our experience in constructing an index type system for JVMLi. We show how our restriction to type terms results in a clear and simple mechanism for typing method invocation. In the sections that follow, we first we present the syntax, the operational semantics and the typing rules in sections 2, 3 and 4, respectively. Then we claim the soundness and safe array access in section 5. Finally we discuss related work and future plan. Some important cases in the proof of theorems are provided in an appendix.

2 Syntax of JVMLi

The syntax of JVMLi is presented in Fig.2. In the rest of paper, we assume that the readers have some level of acquaintance of the JVM [8].

2.1 Values

Values in JVMLi are integers and arrays. Only integer arrays are allowed for simplicity. These arrays are allocated on the heap using the `newarray` instruction. We represent the heap as a collection of arrays, each of them characterized by a base address and a region of memory. As JVML, JVMLi is a stack-based language. Operands and intermediate results are stored in a stack or in one of the local variables. Parameter passing is done through local variables. An operand stack and a set of local variables compose a frame. Each method invocation creates a new frame on the frame stack.

2.2 JVMLi Program Structure

To keep the presentation focused, we consider a subset of the JVM that includes basic memory access instructions (load/store), simple arithmetic, array creation and element access, control instructions, and method invocation and return. The operands allowed are immediate (constant) values, values from the local variable set, values in the current method's operand stack, and values from the heap.

Code is presented as a series of labeled basic blocks. Each block has a single entry. Control operations (jumps) can only target labeled blocks. The order of blocks is significant. If control passes from the end of a basic block it enters the next block in the series unless a control instruction says otherwise. Programs are presented as a set of methods, each of which is a labeled sequence of blocks that is annotated with a type. The only form of method invocation is `invoke`. The only mechanisms to return from a method are `ireturn` or `areturn`.

2.3 Types and Annotations

The type and annotation system is used to statically characterize the potential dynamic behavior of the abstract machine when the program is executed. Intuitively it may be useful to think of the "code" presented as a flow chart and the "types" as Floyd-style annotations on the control arrows in the flowchart that characterize the potential state of the abstract machine whenever that control point is reached. To motivate the details of the type system, it is necessary to mention that the abstract machine state will be characterized by the contents of the heap, the values of the local variables, the operand stack, the program counter, and a frame stack.

$$
\begin{array}{lcl}
B1 & = & b1 : (\cdot.S, r0 : int(i))\{\texttt{ipush } 0;\ \texttt{iload } 0;\ \texttt{isub };\texttt{ifne } nonzero)\} \\
B2 & = & b2 : (\cdot.S, r0 : int(i))\{\texttt{ipush } 0;\ \texttt{ireturn};\} \\
B3 & = & nonzero : (i > 0.S, r0 : int(i)).\{\texttt{iload } 0;\ \texttt{ipush } 1;\ \texttt{iload } 0;\ \texttt{isub};\ \texttt{invoke } m;\} \\
B4 & = & b4 : (i > 0, x >= 0.int(x) :: int(i) :: S, r0 : int(i))\{\texttt{iadd }; \ \texttt{ireturn};\} \\
m & = & code[i : int, i >= 0](int(i) \to j : int, j >= 0.int(j))\{B1; B2; B3; B4\} \\
p & = & \{m\}
\end{array}
$$

Figure 1: A method computing Σi in JVMLi

The presentation of the syntax of the types and annotations includes that of index expressions, index contexts, value types and state types, and function types.

The syntactic phylum of index expressions contains integer constants, variables, and linear arithmetic expressions. The index predicates are predicate logic terms built from equalities and inequalities of index expressions. An index context is a collection of the sort assignment to index variables and the predicates over some index objects. The index contexts are all universally quantified in JVMLi. (Existential quantification is useful for index-typing a number of functions, as pointed out in [23]. We have implemented existential quantification but do not present it in this paper.) With a little misuse of terminology, and because that the most important ingredients in an index context are those predicates representing constraints, we will refer to an index context as a constraint set.

The phylum of value types, τ, is either *int* indexed with an index expression, *intarray* indexed with an index expression, or an existentially quantified type. Hence $int(3)$, $int(x)$, and $intarray(x)$ are all potential types.

The phylum of stack types (that is, types assigned to stacks), S, characterizes the current stack frame. It can either be empty, characterized by a variable, or characterized by a stack type extended with a value type. Similarly, the phylum of local variable (or register) types is given by an association of variable names (register numbers) with value types.

The typing assertions embedded in programs are triples made up of the stack type, local variable typing, and the constraint set characterizing the required relationships between the index variables. Such triples are called state types. In addition, the inference rules exploit a related quadruple that extends the state type with a return type. The return type characterizes the constraints on values returned from a method invocation.

The most interesting aspect of the type system of JVMLi is the treatment of methods. Before refining the types with indexes, methods in JVMLi are essentially monomorphic first-order functions. However when refined by the index types these monomorphic functions become polymorhpic. For example, a `sum` method that adds two numbers would have the mono type indicating it mapped a pair of ints to an int before refinement, but would have the highly polymorhpic type indicating that for every x and y it maps an $\mathrm{int}(x)$ and an $\mathrm{int}(y)$ to an $\mathrm{int}(x+y)$ after refinement. (Technically, this is actually $\forall x, y, z.(\mathrm{int}(x), \mathrm{int}(y)) \to \mathrm{int}(z).z = x + y$.)

As with any polymorphic type system, it then becomes critical to account for the scope of the type variables, or, in this case, the index variables. This management is done within the index contexts and is manipulated by substitutions that occur explicitly in the type rules and type coercion judgments. We use different subscripts to indicate the scopes: ϕ_f for the context at the method level, ϕ_b for the context whose scope is the current basic block and ϕ for combining the two together.

An example method in JVMLi is listed in Fig. 1. It sums up integers up to the arguments. The precondition to the method is that the argument must be non-negative, which is adopted only for illustrative reasons.

In the next section, we present the dynamic semantics of JVMLi in terms of abstract machine state transformer. Then in Section 4, we go into the type realm, discussing type-checking rules.

values	v	$::=$	$i \mid a$
local var values	r	$::=$	$\{r_1 : v_1, \ldots, r_n : v_n\}$
int values	i		
int array values	a		
stack values	s	$::=$	$v :: s \mid \cdot$
heap values	h	$::=$	$\{a_1 : v_{11}, \ldots, v_{1n},$
			$\ldots,$
			$a_m : v_{m1}, \ldots, v_{mk}\}$
frame values	f	$::=$	$(h, r, s, pc, f) \mid \cdot$
state types	σ	$::=$	$\phi.(R, S)$
assertion types	σ_a	$::=$	$\phi.(R, S, \tau_r)$
local variable types	R	$::=$	$[r_0 : \tau_0, \ldots, r_{n_r-1} : \tau_{n_r-1}]$
stack types variables	ρ		
stack types	S	$::=$	$[] \mid \rho \mid \tau :: S$
value types	τ	$::=$	$\mathtt{int}(m) \mid \mathtt{intarray}(m)$
return types	τ_r	$::=$	$\tau \mid \phi.\tau_r$
type erasure	ϵ	$::=$	$\mathtt{intarray} \mid \mathtt{int}$
index contexts	ϕ	$::=$	$\cdot \mid \phi, P \mid \phi, m : \kappa$
index sorts	κ	$::=$	int
index variables	m, n		
int constant	c		
index expression	x, y	$::=$	$c \mid m \mid x + y \mid x - y \mid x * y$
index predicates	P	$::=$	$x \leq y \mid x = y \mid ... \mid P_1 \wedge P_2 \mid P_1 \vee P_2 ...$
local variables	r	$::=$	$r_0, \ldots, r_{n-1}$
instructions	ins	$::=$	$aop \mid bop\ l \mid$
			$\mathtt{iload}\ r \mid \mathtt{istore}\ r \mid \mathtt{astore}\ r \mid \mathtt{aload}\ r \mid$
			$\mathtt{iaload} \mid \mathtt{iastore} \mid \mathtt{ipush}\ x \mid$
			$\mathtt{jmp}\ l \mid \mathtt{newarray} \mid \mathtt{halt} \mid$
			$\mathtt{invoke}\ m \mid \mathtt{ireturn} \mid \mathtt{areturn}$
instruction sequences	I	$::=$	$ins \mid ins; I \mid B$
arithmetic ops	aop	$::=$	$\mathtt{iadd} \mid \mathtt{isub} \mid \mathtt{imul}$
branch ops	bop	$::=$	$\mathtt{ifeq} \mid \mathtt{ifneq} \mid \mathtt{iflt} \mid \mathtt{iflte} \mid \mathtt{ifgt} \mid \mathtt{ifgte}$
labels	l		
basic blocks	B	$::=$	$l : \sigma.\{\text{ins}\}$
methods	M	$::=$	$\mathtt{code}[\phi](\tau_1, \ldots, \tau_n \rightarrow \tau_r)\{B_1; \ldots; B_n\}$
programs	P	$::=$	$\{M\}$

Figure 2: Syntax for $JVML_i$

Current Frame Value	New Frame Value
h,r,v::s,(`newarray`;I),fs	$h[a:0_0,\ldots,0_{v-1}]$,r,$a::s$,I,fs
h,r,$i_1::i_2::s$,(aop;I),fs	h,r,$(i_1\ aop\ i_2)::s$,I,fs
h,r,i::s,(`ifeq` l;I),fs	when $i=0$ (h,r,s,(l:I'),fs) when $i\neq 0$ (h,r,s,I,fs)
h,r,$v_0::\ldots::v_{n-1}::s$,(`invoke` l_m;I),fs	h,$\{r_0:v_0,\ldots,r_{n-1}:v_{n-1}\}$, $\cdot$,$l_m:I'$,(h,r,s,I,fs)
h,r,$v::s$,(`ireturn`;I),(h',r',s',I',fs') h,r,$v::s$,(`areturn`;I),(h',r',s',I',fs')	h,r',v::s',I',fs'
h,$\{\ldots,r_i{:}v_i,\ldots\}$,s,(`iload`$i$;I),fs h,$\{\ldots,r_i{:}v_i,\ldots\}$,s,(`aload`$i$;I),fs	h,r,$v_i::s$, I, fs
h,r,$v_i::s$,(`iastore`i;I),fs h,r,$v_i::s$,(`aastore` i;I),fs	h,$r[r_i:v_i]$,s,I,fs
$h(v_1:(\ldots,v_2:v,\ldots))$,r,$v_1::v_2::s$,(`iaload`;I),fs	h,r, v::s, I,fs
h,r,$v::v_1::v_2::s$,(`iastore`;I),fs	$h(v_1:(\ldots,v_2:v,\ldots))$,r,s,I,fs
h,r,s,(`ipush`i;I),fs	h,r,i::s,I,fs

(h,r,s,I,fs) is the abstract machine state. Components represent the heap, the local variable set, the operand stack, the current instruction sequence and the frame stack, respectively.

Figure 3: Operational Semantics

3 Operational Semantics of JVMLi

Fig. 3.2 presents the essential elements of the operational semantics of JVMLi. It is expressed as a transition system on configurations. A configuration is given by an instruction sequence, an operand stack, a set of local variables, a heap, and a frame stack. The transition relation is written $\rightarrow$. Except as noted the operational semantics agrees with the standard semantics for the JVM. A few cases of the semantics are discussed below.

3.1 Array Access

JVMLi's counterpart of JVML array access instructions are `iaload` and `iastore`. The former pops the base address and the index of an array from the operand stack, fetches the corresponding array element, and pushes it to the stack. The latter gets the base address, the index, and the value to be assigned from the operand stack. It then updates the heap accordingly.

The operational semantics for the array access instructions are defined only when the index is within the bounds of the array. If the index is out of bounds, the computation"gets stuck." This technical property is exploited in the statement of Theorem 5.1 which shows that type correct computations do not "get stuck." In a standard JVM implementation, out of bounds array accesses may be attempted. The JVM semantics specify that an `IndexOutOfBound` exception should be raised in that case. Our intension in this paper is that a well-typed program will not attempt any out-of-bound array accesses. Therefore, there is no need to check the array index dynamically.

3.2 Method Invocation

The rule regarding `invoke` essentially says, when we call a method m, we create a new frame with an empty stack, initialize the local variables set to the arguments on the operand stack of the caller and advance the program counter to the first instruction of the callee.

In JVMLi, method return is done by one of the two instructions: `ireturn` and `areturn`, returning an integer or an address respectively. Note that the return instruction discards the operand stack and local variable set.

4 Typing Rules

The type system describes formally the analysis performed by the receiver to validate the embedded assertions in the code and to assure that the program can be executed safely. It is presented as an inference system defining two basic judgments: a code sequence is safe, written $\phi.(R, S, \tau) \vdash I$ and defined in Fig. 4, and a constraint set implies a property P, written $\phi \models P$. Properties can be either index predicates, such as $\phi \models i < a$, or typing coercion relations induced by index predicates. These are explained in Section 4.3.

4.1 Type Rule

The most direct way to understand the type rules is to think of them as the predicate transformers on the Floyd-style assertions about the potential states of the abstract machine at control points during execution. The basic judgment of the system, $\phi.(R, S, \tau) \vdash I$, indicates that if control reaches the beginning of instruction sequence I in a state satisfying $\phi.(R, S)$ then the execution will continue without a type error (which in this case includes out of bounds array access). The τ component of the judgment specifies the index-types on values to be returned from a method.

The rules can be divided into three categories: information accumulation rules, control rules, and protection rules. The information accumulation rules extend the constraint set, but do not invoke the constraint solver. The control rules invoke the solver to verify that assertions at the entry points to basic blocks are established. The protection rules invoke the solver to establish the safety of a particular operation.

The rule for `iadd` is the paradigmatic example of an information accumulation rule. It asserts that if the stack shape is $int(i) :: int(j) :: S$ and constraint set ϕ holds before the instruction then the stack shape will be $\text{int}(k) :: S$ and the constraint set will include the equation $k = i + j$ after the instruction. This is written formally:

$$\frac{\phi, k : int, k = i + j.(R, int(k) :: S, \tau_r) \vdash I}{\phi.(R, int(i) :: int(j) :: S, \tau_r) \vdash iadd; I} \text{ (type-iadd)}$$

Note that the precondition of the code fragment including the `iadd` instruction is in the consequent of the inference rule and that the assertion describing the program point after the instruction is expressed in the antecedent.

Control instructions precede the entry of basic blocks. The control instructions must invoke the constraint checker to verify that the conditions in the annotations at the entry of basic blocks are satisfied. Essentially, the constraint checker must verify that the strongest postcondition calculated by the type inference rules for the block containing the control instruction entails the precondition of the block to which control is transfered. This is demonstrated by the rule for `ifeq`.

$$\frac{\phi_f \models (\phi_b, i = 0).(R, S) \subseteq \phi_l.\Gamma_l \quad \phi_f \models (\phi_b, i \neq 0).(R, S) \subseteq \phi_{l'}.\Gamma_{l'}}{\phi.(R, int(i) :: S, \tau_r) \vdash \texttt{ifeq}\ \text{l}; (l' : (\phi_{l'}.\Gamma_{l'})\{I\})} \text{ (type-ifeq)}$$

where $\phi_l.\Gamma_l$ is the annotation at label l.

Type rules for array access are categorized as protection rules because they invoke the constraint solver to verify that the constraint set inferred at this program point implies that an operation is safe according to the security policy. The rules for `iaload` and `iastore` are protection rules. The `iaload` rule is:

$$\frac{\phi, k : int.(R, int(k) :: S, \tau_r) \vdash I \quad \phi \models i < a \quad k \text{ is new}}{\phi.(R, int(i) :: intarray(a) :: S, \tau_r) \vdash \texttt{iaload}; I} \text{ (type-iaload)}$$

This rule invokes the constraint solver to establish $\phi \models i < a$. If this is established then the array access is provably safe.

$$\frac{\phi, \phi_r[\theta].(R, \tau_{ret}[\theta] :: S, \tau_r) \vdash I \quad \forall i = 0, \ldots, n-1.\ \phi. \models \tau_i \subseteq \tau_i'[\theta]}{\phi.(R, \tau_1, \ldots, \tau_n :: S, \tau_r) \vdash \texttt{invoke}\ m; I} \text{ (type-invoke)}$$

where $M_p(m) = code(\phi')(\tau_0', \ldots, \tau_{n-1}' \rightarrow (\phi_r.\tau_{ret}))\{B\}$

$$\frac{\phi_f \models (\phi_b, i = 0).(R, S) \subseteq \phi_l.\Gamma_l \quad \phi_f \models (\phi_b, i \neq 0).(R, S) \subseteq \phi_{l'}.\Gamma_{l'}}{\phi.(R, int(i) :: S, \tau_r) \vdash \texttt{ifeq}\ \mathrm{l}; (l' : (\phi_{l'}.\Gamma_{l'})\{I\})} \text{ (type-ifeq)}$$

where $\phi_l.\Gamma_l$ is the annotation at label l.

$$\frac{\phi, k : int, k = i + j.(R, int(k) :: S, \tau_r) \vdash I \quad k \text{ is new}}{\phi.(R, int(i) :: int(j) :: S, \tau_r) \vdash \texttt{iadd}; I} \text{ (type-iadd)}$$

$$\frac{\phi.(R, \tau_i :: S, \tau_r) \vdash I \quad \|\tau_i\| = int}{\phi.(R[r_i : \tau_i], S, \tau_r) \vdash \texttt{iload}\ i; I} \text{ (type-iload)}$$

$$\frac{\phi.(R, \tau_i :: S, \tau_r) \vdash I \quad \|\tau_i\| = intarray}{\phi.(R[r_i : \tau_i], S, \tau_r) \vdash \texttt{aload}\ i; I} \text{ (type-aload)}$$

$$\frac{\phi.(R[r_i \mapsto \tau_i], S, \tau_r) \vdash I \quad \|\tau_i\| = int}{\phi.(R, \tau_i :: S, \tau_r) \vdash \texttt{istore}\ i; I} \text{ (type-istore)}$$

$$\frac{\phi.(R[r_i \mapsto \tau_i], S, \tau_r) \vdash I \quad \|\tau_i\| = intarray}{\phi.(R, \tau_i :: S, \tau_r) \vdash \texttt{astore}\ i; I} \text{ (type-astore)}$$

$$\frac{\phi \models i < a \quad \phi.(R, S, \tau_r) \vdash I \quad \|\tau_v\| = int}{\phi.(R, \tau_v :: int(i) :: intarray(a) :: S, \tau_r) \vdash \texttt{iastore}; I} \text{ (type-iastore)}$$

$$\frac{\phi, k : int.(R, int(k) :: S, \tau_r) \vdash I \quad \phi \models i < a \quad k \text{ is new}}{\phi.(R, int(i) :: intarray(a) :: S, \tau_r) \vdash \texttt{iaload}; I} \text{ (type-iaload)}$$

$$\frac{\phi, k : int, k = c.(R, int(k) :: S, \tau_r) \vdash I \quad k \text{ is new}}{\phi.(R, S, \tau_r) \vdash \texttt{ipush}\ c; I} \text{ (type-ipush)}$$

$$\frac{\phi \models \tau \subseteq \tau_r \quad \|\tau\| = int}{\phi.(R, \tau :: S, \tau_r) \vdash \texttt{ireturn}; I} \text{ (type-ireturn)}$$

$$\frac{\phi \models \tau \subseteq \tau_r \quad \|\tau\| = intarray}{\phi.(R, \tau :: S, \tau_r) \vdash \texttt{areturn}; I} \text{ (type-areturn)}$$

Figure 4: Typing rules: instructions

$$\frac{\phi \models x = y}{\phi \models int(x) \subseteq int(y)}$$

$$\frac{\phi \models x = y}{\phi \models intarray(x) \subseteq intarray(y)}$$

$$\frac{\phi \vdash \theta : \phi_2 \quad \phi \models \tau_1 \subseteq \tau_2[\theta]}{\phi_1 \models \tau_1 \subseteq \exists \phi_2 . \tau_2} \text{ (coerce-ret-type)}$$

$$\frac{\phi_1 \vdash \theta : \phi_2 \quad \phi_1.(R, S, \tau) \models (R_2, S_2, \tau)[\theta]}{\phi_1.(R, S, \tau) \models \phi_2.(R_2, S_2, \tau)} \text{ (coerce-substitution)}$$

$$\frac{\phi.(R, S) \models R' \quad \phi.(R, S) \models S'}{\phi.(R, S, \tau) \models (R', S', \tau)} \text{ (coerce-state)}$$

$$\frac{}{\phi.(R, S, \tau) \models \rho} \text{ (coerce-stack-var)}$$

$$\frac{\phi.(R_1, S, \tau) \models S_2 \quad \phi \models \tau \subseteq \tau_2}{\phi.(\tau :: R_1, S, \tau) \models (\tau_2 :: S_2)} \text{ (coerce-stack)}$$

$$\frac{\forall n.(\phi \models R(n) \subseteq R'(n))}{\phi.(R, S, \tau) \models R'} \text{ (coerce-localv)}$$

$$\frac{\phi_1, \phi_2.(R, S, \tau) \models \phi_3.(R_1, S_1, \tau_1)}{\phi_1 \models \phi_2.(R, S, \tau) \subseteq \phi_3.(R_1, S_1, \tau_1)} \text{ (coerce-quan)}$$

Figure 5: type coercion judgment

$$\Phi ::= P \mid P \supset \Phi \mid \forall a.\Phi$$

Figure 6: constraint language

The type erasure rules used in the type rules is defined as follows.

$$\begin{array}{rcl} \|int(x)\| & = & int \\ \|intarray(x)\| & = & intarray \end{array}$$

To type-check a JVMLi program, we type-check the instruction sequence of every method. In a successful derivation, we assign the assertion type representing the precondition to a program point.

4.2 Method Invocation

Method invocation is checked by a control rule. This rule implements index polymorphism by specializing the method type with the substitution θ. The substitution maps the index variables of the formal arguments to the index variables of the actual arguments.

$$\frac{\phi, \phi_r[\theta].(R, \tau_{ret}[\theta] :: S, \tau_r) \vdash I \quad \forall i = 0, \ldots, n-1.\ \phi. \models \tau_i \subseteq \tau_i'[\theta]}{\phi.(R, \tau_1, \ldots, \tau_n :: S, \tau_r) \vdash \texttt{invoke}\ m; I}\ \text{(type-invoke)}$$

where $M_p(m) = code(\phi')(\tau_0', \ldots, \tau_{n-1}' \rightarrow (\phi_r.\tau_{ret}))\{B\}$

4.3 Constraint Satisfaction

Both the control and protection rules in the type system generate verification conditions that are discharged by a constraint satisfaction engine. The queries generated are of two forms: $\phi \models P$, where P is a predicate over integers, and $\phi \models \tau_1 \subseteq \tau_2$, where $\tau_1 \subseteq \tau_2$ is an induced subtype relation derived from integer predicates.

Type coercion rules are listed in Fig. 5. Intuitively, the left hand side of $\subseteq$ is a type with "stronger" properties; it can be coerced into a type with "weaker" properties. Substitutions, symbolized as θ defined in these rules are standard. We need only to mention that $\phi \vdash \theta : \phi_1$ means a substitution of ϕ to ϕ_1.

The type coercion rules, in most cases, end up with judgments of the form $\phi \models Pred$, where $Pred$ is a set of predicates over integers. In the absence of any "qualified" types, where ϕ contains only type assignments, it is trivial to make judgments of this form. However, when index types are involved, a constraint domain decision procedure, called the constraint solver, must be called to make the judgment. In our case, a linear programming procedure is used to determine whether these predicates hold.

Starting from judgment $\phi \models Pred$, we define the following procedure to generate an expression, denoted by Φ, that is accepted by the constraint solver. The $\Rightarrow$ sign defines one step from judgment $\phi \models Pred$ to the constraint language accepted by the solver.

4.4 Implementation

We have implemented the type-checking procedure in Haskell. The constraint solver is based on branch-and-bound integer programming and is written in C. Following the interface of so-called "duck-logic" [1], we integrate the constraint solver with the Haskell type-checker using HDirect [2].

$$
\begin{array}{l}
\cdot \models \Phi \Rightarrow \Phi \\
\phi, a : int \models \Phi \Rightarrow \phi \models (\forall a : int)\Phi \\
(\phi, P) \models \Phi \Rightarrow \phi \models P \supset \Phi
\end{array}
$$

Figure 7: transformation to constraints

$Instruction$	$post-condition$	$comments$
`ipush` 0;	$i >= 0, k = 0.(int(k) :: S, r0 : int(i))$	
`iload` 0;	$i >= 0, k = 0.(int(i) :: int(k) :: S, r0 : int(i))$	
`isub`;	$i >= 0, k = 0, l = i - k.(int(l) :: S, r0 : int(i))$	
`ifne` $nonzero$;		
`ipush` 0;	$i = 0, m = 0.(S, r0 : int(i))$	
`ireturn`;		$m :: (i >= 0)int(i)$ $\rightarrow$ $j : int, j > 0.int(j)$ prove $m > 0$ note: $j \mapsto m$
`iload` 0;	$i > 0.(int(i) :: S, r0 : int(i))$	
`ipush` 1;	$i > 0, n = 1.(int(n) :: int(i) :: S, r0 : int(i))$	
`iload` 0;	$i > 0, n = 1.(int(i) :: int(n) :: int(i) :: S, r0 : int(i))$	
`isub`;	$i > 0; n = 1; o = i - n.(int(o) :: int(i) :: S, r0 : int(i))$	
`invoke` m;	$i > 0, p >= 0.(int(p) :: int(i) :: S, r0 : int(i))$	
`iadd`;	$i > 0, p >= 0, q = p + i.(int(q) :: S, r0 : int(i))$	
`ireturn`;		

Figure 8: Postconditions for the method in Fig.1

In Fig. 1 we demonstrated an example of a JVMLi method, which sums up integers 1 through n, where n is the argument. The post-conditions assigned to the instructions of this method are shown in Fig. 8.

5 Soundness

To state the soundness and subject reduction properties of the system we need to formally characterize what it means for a configuration of the abstract machine to correspond to a type at a program point and we need to introduce a mechanism to refer to the inferred type at all program points.

The "is-of-type" relation defined below relates abstract machine states to state types.

$$\frac{P(i)}{x : int, ..., P(x) \models i : int(x)} \ \textbf{(value-type-inti)}$$

$$\frac{\text{a is an integer array of length len} \quad P(len)}{n : int, ..., P(n) \models a : intarray(n)} \ \textbf{(value-type-arrayn)}$$

$$\frac{\phi \models ia : \tau[a \mapsto x] \quad \phi \vdash P(x)}{\phi \models ia : P(a).\tau} \ \textbf{(value-type-exist)}$$

$$\frac{}{\phi \models [] : \bot} \textbf{(stack-type-empty)}$$

$$\frac{\phi \models s : S \quad \phi \models ia : \tau}{\phi \models (ia \cdot s) : (\tau \cdot S)} \textbf{(stack-type-rest)}$$

$$\frac{\forall i \in N. \phi \models r[i] : R[i]}{\phi \models r : R} \textbf{(localv-type)}$$

$$\frac{\phi \models s : S \quad \phi \models v : V}{\phi \models (r, s) : (R, S)} \textbf{(stack-localv-type)}$$

To state the subject reduction property we must extend the map from labels to assertions to every program point by defining it to be the strongest postcondition deduced by applications of the inference rules from the assertion at the entry point to the enclosing basic block.

Before proving soundness two properties are proved about stack types and local variable types, respectively. These properties show that type refinements consistent with the constraints are sound with respect to the is-of-type relation defined above.

Proposition 5.1 *For all stacks s, environments ϕ, local variable types R, return types τ_r , if $\phi \models s : S_1$ and $\phi.(R, S_1, \tau_r) \models S_2$, then $\phi \models s : S_2$.*

Proposition 5.2 *If $\phi \models r : R_1$ and $\phi.(R_1, S, \tau_r) \models R_1$, then $\phi \models r : R_2$,*

Subject reduction typically characterizes that the type of a term is invariant under reduction. In this context we do not have a traditional reduction semantics. However, there is a key invariant that is preserved by the transition relation in the abstract machine. That property is that the soundness of the relationship between program points and state type assertions (given by D) is preserved. This property is stated formally as follows:

Lemma 5.1 (One-step Lemma (Subject Reduction)) *Suppose a well-typed program P goes one step from (h, s, v, I, fs) to (h', s', v', I', fs'); suppose $D(I) = \phi.(R_1, S_1)$ and $D(I') = \phi'.(R_2, S_2)$ are the state-type assigned to I and I', respectively; suppose $\phi \models (r, s) : (R_1, S_1)$,then $\phi' \models (r', s') : (R_2, S_2)$.*

Note that the subject reduction stated above also implies the annotated method type is correct for a well-typed program. This is clear in its proof. Complete proofs of these properties may be found in the companion technical report. In the Appendix, we treat several cases of the subject reduction lemma.

In addition to showing that every step taken preserves the type invariant, we must also show that the machine never "gets stuck" to argue formally that no unsafe array references are attempted by a type correct program. That property is stated formally:

Theorem 5.1 *If a program is well-typed, and assume for empty stack s_0, empty local variable set r_0, the instruction sequence of a top-level method (main) I_m, and an initial frame stack fs_0, $(h_0, s_0, v_0, I_m, fs_0) \rightarrow^* (h, s, v, I, fs)$, then either there exists (h', s', v', I', fs') so that $(h, s, v, I, fs) \rightarrow (h', s', v', I', fs')$, or the current instruction is halt.*

where the $\rightarrow^*$ means applying the operational semantic rules one or more times.

Theorem 5.2 *A well-typed JVMLi will not make out-of-bound array access.*

Another result is that the annotations in a well-typed program is verified by the type-checking. It comes from the following theorem and the soundness.

Theorem 5.3 *If a program is well-typed, then for every basic block $l : \phi_b.(S, R).I$ in method $\texttt{code}[\phi_f](\tau_1, \ldots, \tau_n \rightarrow \phi_r.\tau_r)\{B_1; \ldots; B_n\}$, $D(I) = \phi_r, \phi_b.(S, R, \phi_r.\tau_r)$*

6 Related Work

Program analysis techniques statically compute reliable approximate information about the dynamic behavior of a program [12]. Eliminating unnecessary array bound checks is a standard application of program analysis [4, 5]. Our work can be regarded as one way to embed such an analysis in a type system. In particular, the handling of constraints in our type system resembles that in a constraint based analysis method. The difference between our type-checking process and program analysis is the use of the type annotations, which help to draw stronger conclusions about program behavior.

Type inference in a functional language is similar to the value inference in a logic programming language. An obvious example is the algorithm used in Hindley-Milner type inference is the algorithm used in a prolog interpreter [15]. In fact, Xi's original work on DML relates to the constraints in CLP. Recent work by Sulzmann et al proposes a framework to form dependent type system from a constraint system that satisfies certain properties [17, 18, 19]. The attractive part of this plan is that we can always expect a decidable system.

Typing the JVM bytecode language has been done in various research projects [16, 3, 13]. We conjecture that some of these type systems can be put into an index type framework similar to the one presented in this paper. We also conjecture that we can use index types to characterize the shared variables in JVM to deal with the JVM concurrency model. However, we may have to construct a different index type system for a different analysis. Our future plan is to find a way to compose these type systems in a systematic way, as is described in Chen and Jones' recent research.

7 Acknowledgment

We thank Hongwei Xi, who inspired and guided the early work. He also observed the problem with quantifying index context to blocks. We also thank Mark Jones, Nancy Day, Tim Sheard, John Peterson, and Martin Sulzmann for many helpful discussions. We are particularly grateful to Lois Delcambre and the other readers of the authors' research proficiency exam. We thank the entire Pacsoft group at Oregon Graduate Institute for their support and encouragement. Finally, we thank the three anonymous referees who gave many useful comments and suggestions which have helped improve the paper significantly.

This research is partially supported by Air Force Material Command, the National Science Foundation and DARPA.

References

[1] Day, N. A., Launchbury, J., and Lewis, J. Logical Abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop* (October 1999), Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28.

[2] Finne, S. Haskell Direct, 2000. Webpage at: http://www.dcs.gla.ac.uk/fp/software/hdirect/.

[3] Freund, S., and Michell, J. A Type System for Object Initialization in the Java Bytecode Language. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application* (1998), pp. 310–327.

[4] Gupta, R. Optimizing Array Bound Checks Using Flow Analysis. In *ACM Letters on Programming Languages and Systems 2(1-14)* (1994), pp. 135–150.

[5] Kolte, P., and Wolfe, M. Elimination of Redundant Array Subscript Checks. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (1995), ACM Press.

[6] Kozen, D. Efficient Code Certification, January 1998. Technical Report 98-1661, Computer Science Department, Cornell University.

[7] Kozen, D. *Language-Based Security*, vol. 1672. Springer-Verlag, September 1999.

[8] Lindholm, T., and Yelin, F. *Java Virtual Machine Specification.* Andison-Wesley, 1996.

[9] Morrisett, G., Crary, K., Glew, N., and Walker, D. Stack-based Typed Assembly Language. In *Proceedings of Workshop on Types in Compilation* (1998).

[10] Morrisett, G., Walker, D., Crary, K., and Glew, N. From System F to Typed Assembly Language. In *Proceedings of ACM Symposium On Principles of Programming Languages* (1998), pp. 85–97.

[11] Necula, G. *Proof-carrying Code.* ACM Press, 1997, pp. 106–119.

[12] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis.* Springer, October 1999.

[13] O'Callahan, R. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programmming Languages* (1999), pp. 70–78.

[14] Okasaki, C. *Purely Functional Data Strucutures.* Cambridge University Press, 1998.

[15] Schmidt, D. *The Structure of Typed Programming Languages.* The MIT Press, 1994.

[16] Stata, R., and Abadi, M. A Type System for Java Bytecode Subroutines. In *Proceedings of ACM Symposium on Principles of Programming Languages* (1998), pp. 149–160.

[17] Sulzmann, M., March 2000. Personal Discussion.

[18] Sulzmann, M. *A General Framework For Hindley-Milner Type System with Constraints.* PhD thesis, Yale University, 2000. forthcoming.

[19] Sulzmann, M., Odersky, M., and Wehr, M. Type Inference with Constraint Types, 1997. Research Report YALEU/DCS/RR-1129.

[20] Xi, H. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998. PhD Thesis, available at http://www.cs.cmu.edu/˜ hwxi/DML/thesis.ps.

[21] Xi, H. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)* (Paris, September 1999), pp. 17–32.

[22] Xi, H., and Pfening, F. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 249–257.

[23] Xi, H., and Pfenning, F. Dependent Types in Practical Programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages* (1999), pp. 214–227.

[24] Xi, H., and Xia, S. Toward Elimination of Array Bound Checks in JVM. In *Proceeding of CASCON 99* (1999), pp. 147–156.

[25] Zenger, C. Index Types. In *Theoretical Computer Science* (1998), vol. 187, Elsevier, pp. 147–165.

[26] Zenger, C. *Indizierte Typen.* PhD thesis, University of Karlsruhe, 1998.

8 Appendix: Part of Soundness Proof

In this appendix, we demonstrate how we prove the soundness theorem by treating several important cases for the subject reduction lemma. Due to the similarities between proof for stack types and local variable types, we only deal with stack types here.

In the following proof, we will refer to the assertion type assigned to an instruction sequence $ins; I$ as D(ins;I). We will use $D(ins; I).s$ and $D(ins; I).lv$ as selectors of stack type and local variable type parts in $D(ins; I)$.

Consider $ins; I$, we do a case split on ins. Since the program goes one step further, ins cannot be $halt$.

First consider the case when I does not begin with a basic block.

- Case 1: ins is `iadd`, `isub`, or `imul`. From the dynamic semantics, we know $\exists s'', x, y.\ s = x :: y :: s'', s' = (x\ op\ y) :: s''$. From the static semantics, $\exists S'', m, n.\ D(ins; I).s = int(m) :: int(n) :: S'', D(I).s = int(m\ op\ n) :: S''$. Given $s : D(I).s$, it follows that $s'' : S''$, $(x\ op\ y) : int(m\ op\ n)$ and then $s' : D(I)$.s.

- Case 2: ins is `iaload` or `iastore`. We take `iastore` as an example. From the dynamic semantics, we know $\exists s'', x, y, a.\ s = x :: y :: a :: s'' \wedge s' = s''$ From the static semantics, we have $\exists S'', i, n : i < n.D(ins; I).s = int :: int(i) :: intarray(n) :: S'', D(I).s = S''$ Since the program runs a step, then from the dynamic semantics, the evaluation rule must be defined for (a,y). Therefore $a < y$. It follows that $s'' : S''$ and thus $s' : D(I).s$.

- Case 3: ins= `iload`, `istore`, `aload`, `astore`, `pop`, `ipush`. We take $iload$ as an example. From the dynamic semantics, we know $\exists s'', integerx.\ s = x :: s'' ands' = s''$ From the static semantics, we know $\exists S'', m.D(pc).s = int(m) :: S'', D(pc{+}1).s = S''$ It follows directly $s'' : S''$.

- Case 4: ins = `jump`, `ifeq`, `ifne`, `ifgt`, `invoke`, `ireturn`, `areturn`. Since they are control instructions. We discuss them in the next case

Now we consider the next program locations are basic block entrances. Again we do case split on instructions.

- Case 1: ins = `iaload`,`iastore`,`iload`,`istore`,`aload`, `astore`,`pop`, `ipush`,`isub`,`iadd`,or `imul`. We take `iastore` as an example. From the dynamic semantics, we have $\exists s'', x, y, a.\ s = x :: y :::: as'' \wedge s' = s''$ From the static semantics, we have $\exists S'', i, n, i < n.D(ins; I).s = int :: int(i) :: intarray(n) :: S''$ and that $D(I').s \subseteq D(I).s$, where $D(I')$ is the post condition of the instruction. We know that $s' : D'(I)$. Using proposition 1, $s' : D(I)$.

- Case 2: ins = `jump` l. From the dynamic semantics, $s' = s$. From the static semantics, $D(ins; I).s \subseteq D(I).s$. Following Proposition 1, we have $s' : D(I).s$.

- Case 3: ins = `ifgt`, `ifeq`, `ifne`,... Whichever branch is entered during execution, the two cases are similar to each other and to Case 2.

- Case 4: ins = `invoke`, `ireturn`, `areturn`. If ins is `invoke`, then the next instruction, according to dynamic semantics, is the first instruction of the callee. Since we require the annotation at the beginning of a method to have an empty stack assignment, the conclusion trivially holds. If ins is a return instruction, because the machine runs a step forward, then the fs must not be empty, and since only `invoke` can push a frame onto fs, so fs's top element must be a frame with an instruction I', where I' is the after where the `invoke` is called. Therefore, $s' = v :: fs.s$, where v is return value. As `ireturn` is well-typed, $v : P(X, k).int(k)$, where X represents the index variables for arguments. In other words, $P(X, v)$ holds (in integer domain). So for some Y, $P(Y, v)$ also holds. From static sematics, the top element of D(I').s is of type $P(Y, k').int(k)$, therefore v is of this type too. The existence of an early execution of invoke guarantee that the rest of stack keeps the type.