Validation of Stack Effects in Java Bytecode

Jaanus Pöial
Institute of Computer Science
University of Tartu, Estonia
e-mail: jaanus@cs.ut.ee

February 21, 1997

Abstract

The Java language is widely used in networked world to distribute the platform independent software pieces in form of bytecode for Java Virtual Machine (JVM). Java compilers put a lot of emphasis on early checking for possible problems, runtime checking, and eliminating situations that are error prone. On the other hand, the problem of independent validation of Java bytecode "at receivers end" is still an actual issue for several reasons – security, native code generation/optimisation, etc.

Current paper is an attempt to apply the stack effect calculus (originally designed by the author for the Forth programming language) to JVM operand stack manipulations using interface descriptions of JVM instructions. This approach may be used for bytecode validation against illegal stack manipulations performing some analysis of the code instead of executing it. Also it is possible to apply this theory to the bytecode compiler itself to check that no illegal programs in sense of operand stack usage will be generated.

Stack effects determine some syntactic rules for stack machine programs. In the last section of the paper the relationship between stack effects and syntactic equations (general rewriting rules) is investigated for one particular case.

1 Stack effects

The Java virtual machine [JVM95] is an abstract device for running Java bytecode – the new more or less "common" form of distributing software over the net. This code may be interpreted by some JVM engine or converted to native machine code in which case the programs run considerably faster. The quality of the bytecode itself is also very important if portability to wide range of computers is considered.

JVM is a stack machine and it is quite natural to apply some general techniques of compilation and optimisation for stack machines to the bytecode. There are many differences between Java and earlier approaches (e.g. UNCOL, p-code, DIANA, Forth language/machine, etc.) but also many aspects in common. In this paper we will concentrate on parameter passing through the operand stack of JVM using the so called stack effect calculus. Stack effect checking is a part of bytecode validation that may help in stage of debugging the compiler or other tools, bytecode-level optimisation, preprocessing programs which come of untrusted source, etc.

The main goal of stack effect calculus is static type checking of stack machine programs (and tools that generate such programs). Types, subtypes, "wildcard" types and rules for calculating resulting stack effects for different constructs have been introduced in [Pöi90], [Pöi91], [StK93] and [Pöi94].

Let us define the basic notation and list some results.

We will consider that stack effects (type signatures) form a polycyclic monoid (introduced in [NiP70]). This holds for simple signatures (uniquely defined stack effects without "wildcards" and subtyping [Pöi90]). This also holds for compound signatures (multiple stack effects [Pöi91]) with essential restrictions. Set of multiple stack effects in general is not even an inverse semigroup but there always exists a subset which is a polycyclic monoid. For simplicity we use the set of simple signatures everywhere because it represents some common properties of all inverse semigroups ([ClP67] introduces the basic theory of semigroups).

Let *types* be denoted by a, b, c ... In real programs these can be `int`, `long`, `float`, `double`, `object`, etc. – we do not need any concrete interpretation yet. Let $\mathcal{T}$ be the set of types.

We will use α, β, γ ... for *type lists*. These are finite sequences of types where the rightmost element corresponds to the top of the stack. The set of such lists is $\mathcal{T}^*$.

A *type clash* appears when some stack operation finds an input argument on the stack which has the unexpected (incompatible) type. We will use the symbol $\emptyset$ for the type clash.

A *stack effect* is a pair of input parameter types and output parameter types. We also consider the type clash as a stack effect. The set of stack effects is defined as follows

$$\mathcal{S} = (\mathcal{T}^* \times \mathcal{T}^*) \cup \{\emptyset\}$$

We use s, t, u ... for stack effects as well as $(\alpha - -\beta)$ for the pair $(\alpha, \beta) \in \mathcal{T}^* \times \mathcal{T}^*$. Sometimes we use indices to express inputs and outputs

$$s = (s_1 -- s_2) \text{ where } s_1, s_2 \in \mathcal{T}^*$$

The *composition* (multiplication) of stack effects is defined as follows

$\forall s \in \mathcal{S} : s\ \emptyset = \emptyset\ s = \emptyset$

$\forall s_1, s_2, t_1, t_2, \alpha, \beta \in \mathcal{T}^* :$

$(s_1 -- s_2)(\alpha s_2 -- t_2) = (\alpha s_1 -- t_2)$

$(s_1 -- \beta t_1)(t_1 -- t_2) = (s_1 -- \beta t_2)$

In all other cases the result will be $\emptyset$.

$\mathbf{1} = (\ \text{--}\ )$ is a unity for this operation.

We have an algebraic structure now which is isomorphic to the polycyclic monoid. It has a unity $\mathbf{1}$, a null element $\emptyset$ and an associative operation of multiplication.

Example 1

Let us calculate some products using the definitions above.

$(a -- bc)(c -- de) = (a -- bde)$

$(ab -- c)(dc -- e) = (dab -- e)$

$(a -- b)(bc -- d) = \emptyset$

$(-- ab)(ab --) = \mathbf{1}$

$(a -- b)(cb - d) = (ca -- d)$

$(ca -- cb)(cb -- d) = (ca -- d)$

Let us define the *inverse element* for any $s \in \mathcal{S}$ in the following way

$s = \emptyset \Rightarrow s^{-1} = \emptyset$, i.e. $\emptyset^{-1} = \emptyset$

$s = (s_1 -- s_2) \Rightarrow s^{-1} = (s_2 -- s_1)$, i.e. $(s_1 -- s_2)^{-1} = (s_2 -- s_1)$

This definition introduces a unique inverse element for each stack effect and allows to define the *partial order relation* $\leq$ as follows

$\emptyset \leq s$ for any $s \in \mathcal{S}$ and

$(\gamma\alpha -- \gamma\beta) \leq (\alpha -- \beta)$ for each $\alpha, \beta, \gamma \in \mathcal{T}^*$

It is equivalent to the classical definition

$$s \leq t \Leftrightarrow st^{-1} = ss^{-1}$$

and for non-zero effects the following equivalence holds

$$(\;-\!-\; s_1)(t_1 \;-\!-\; t_2)(s_2 \;-\!-\;) = \mathbf{1} \Leftrightarrow (s_1 \;-\!-\; s_2) \leq (t_1 \;-\!-\; t_2)$$

All *idempotents* of $\mathcal{S}$, i.e. elements u for which $u = uu$, form a commutative subsemigroup of $\mathcal{S}$ with unity and null element. Non-zero idempotents have a form of $(\alpha \;-\!-\; \alpha)$, where $\alpha \in \mathcal{T}^*$.

Having these basic definitions we can return to the question about multiple stack effects (compound signatures). Subset $\mathcal{M} \subset \mathbf{2}^{\mathcal{S}}$, where $\mathbf{2}^{\mathcal{S}}$ denotes the powerset (set of all subsets) of $\mathcal{S}$, is an inverse semigroup, iff

1) $\sigma, \tau \in \mathcal{M} \Rightarrow \sigma\tau \in \mathcal{M}$ ($\mathcal{M}$ is a subsemigroup),
2) $\forall \sigma \in \mathcal{M}\ \exists \tau \in \mathcal{M} : \ \sigma\tau\sigma = \sigma$ (all elements are *regular*),
3) $\sigma, \tau \in \mathcal{M},\ \sigma\sigma = \sigma,\ \tau\tau = \tau \ \Rightarrow \sigma\tau = \tau\sigma$ (all idempotents commutate).

It is not only a question of how to define $\mathcal{M}$, but also how to define the multiplication (and addition).

2 Validation of stack machine programs

Let us have a set of stack operations Δ. We can build programs by writing sequences of stack operations (let us forget about control transfer instructions at the beginning). The set of all "programs" (including these which make no sense) is Δ^*. Each operation $p \in \Delta$ has a given stack effect $sig(p) \in \mathcal{S}$. Mapping $sig : \Delta^* \to \mathcal{S}$ is defined as homomorphism

$sig(empty\_program) = \mathbf{1}$,
$sig(pq) = sig(p)sig(q)$.

Now it is possible to calculate the stack effect of a given program simply by multiplying stack effects of its parts (notice that we need associativity and homomorphism to do this).

The set Δ and homomorphism sig determine a language of valid programs (programs without type clash)

$$Valid(\Delta, sig) = \{\omega \in \Delta^* : sig(\omega) \neq \emptyset\}$$

In some cases a subset of valid programs without input and output parameters is considered

$$Closed(\Delta, sig) = \{\omega \in \Delta^* : sig(\omega) = \mathbf{1}\}$$

Obviously

$$empty\_program \in Closed \subset Valid \subset \Delta^*$$

There are different ways to treat the correctness of programs with regard to the stack effects.

- Exact matching when the calculated effect has to be equal to the desired one. If the desired effect is $\mathbf{1}$ (no inputs, no outputs) we have a *closed program*.
- "Operational matching" when the desired effect may be less (in sense of our partial order relation) than the calculated one. We see from calculations that the program operates correctly but does not use all (bottom) elements on the stack.
- We do not have any idea about desired effects, we only want to have valid programs.

Example 2

If the calculation gives $sig(p) = (bc -- bd)$ then

p is type correct w.r.t. $(bc -- bd)$ in sense of all definitions.

p is not type correct w.r.t. $(abc -- abd)$ in sense of first definition, but satisfies the second one, because $(abc -- abd) \le (bc -- bd)$. p simply does not use the element a .

p is not type correct w.r.t. $(c -- d)$, but is still valid. The reason here is that we cannot put p into context where for example ac is on the top of the stack. It leads to the type clash between a (in context) and b (actually used by program p). At the same time $(c -- d)$ works well in this context and we will not notice the error.

If the programs are generated by a context free grammar then it is possible to guarantee their type correctness by checking the grammar (see [Pöi90] for the details). We need to bind an inequality to each grammar rule and to solve the system of inequalities in $\mathcal{S}$.

Example 3

Let us define Δ and sig as follows

$sig(\texttt{iconst}) = (-- i)$

$sig(\texttt{fconst}) = (-- f)$

$sig(\texttt{fnewarray}) = (i -- a)$

$sig(\texttt{astore}) = (a --)$

$sig(\texttt{aload}) = (-- a)$

$sig(\texttt{fstore}) = (f --)$

$sig(\texttt{fload}) = (-- f)$

$sig(\texttt{iadd}) = (i \; i -- i)$

$sig(\texttt{fadd}) = (f \; f -- f)$

$sig(\texttt{fastore}) = (a \; i \; f --)$

$sig(\texttt{faload}) = (a \; i -- f)$

$sig(\texttt{arraylength}) = (a -- i)$

The grammar on the left produces the system of inequalities on the right.

		1	$\le$ S
$<S>$	$\to <Stms>$	S	$\le$ Stms
$<Stms>$	$\to <Stm>$	Stms	$\le$ Stm
	$\to <Stms><Stm>$	Stms	$\le$ Stms Stm
$<Stm>$	$\to <A>$ `astore`	Stm	$\le$ A $(a --)$
	$\to <A><I><F>$ `fastore`	Stm	$\le$ A I F $(a \; i \; f --)$
	$\to <F>$ `fstore`	Stm	$\le$ F $(f --)$
$<A>$	$\to <I>$ `fnewarray`	A	$\le$ I $(i -- a)$
	$\to$ `aload`	A	$\le (-- a)$
$<I>$	$\to <Ia>$	I	$\le$ Ia
	$\to <I><Ia>$ `iadd`	I	$\le$ I Ia $(i \; i -- i)$
$<Ia>$	$\to$ `iconst`	Ia	$\le (-- i)$
	$\to <A>$ `arraylength`	Ia	$\le$ A $(a -- i)$
	$\to <I>$	Ia	$\le$ I
$<F>$	$\to <Fa>$	F	$\le$ Fa
	$\to <F><Fa>$ `fadd`	F	$\le$ F Fa $(f \; f -- f)$
$<Fa>$	$\to$ `fconst`	Fa	$\le (-- f)$
	$\to <A><I>$ `faload`	Fa	$\le$ A I $(a \; i -- f)$
	$\to$ `fload`	Fa	$\le (-- f)$
	$\to <F>$	Fa	$\le$ F

This grammar is an output grammar of some syntax directed translation scheme which has a normal reduced input grammar with all the ”syntactic sugar”.

The system has a solution
S = Stms = Stm = $(\ --\ ) = \mathbf{1}$,
A = $(\ --\ a)$,
I = Ia = $(\ --\ i)$ and F = Fa = $(\ --\ f)$.

The grammar produces only closed programs, but not *all* closed programs. For example,

`aload iconst fnewarray astore iconst fconst fastore`

is a closed program not generated by this grammar.

We do not need concrete interpretation of stack operations in this paper but for JVM previous program may look like

```
aload_0
iconst_5
newarray <T_FLOAT>
astore_1
iconst_1
fconst_2
fastore
```

3 Stack effects and syntactic equations

We already defined two non-empty languages $Valid(\Delta, sig)$ and $Closed(\Delta, sig)$. Formally this is enough to define a language, but we need to be convinced in usefulness of such a definition. One the one hand, we can calculate the stack effect of a program to decide whether it belongs to the language (no more than semigroup is needed to do this). On the other hand, we need a lot of algebraic properties of the polycyclic monoid to transform the "stack effect syntax notation" to some other form. That is the reason why we still consider that stack effects form at least an inverse semigroup. In this work we are not interested in (very important) questions of concrete type systems, rules for control transfer, etc. Our interest is concentrated on binding the stack effect calculus to methods of syntax description. We will show that the stack effect calculus and general rewriting rules (syntactic equations) are equivalent in particular cases.

Example 4

Let us have Δ and *sig* of example 3 and the language $Closed(\Delta, sig)$. We will show that the following syntactic equations define $Closed(\Delta, sig)$ identically (the grammar above defined only a *subset* of the language).

(1) *empty_program* = `aload astore`
(2) *empty_program* = `aload iconst fconst fastore`
(3) *empty_program* = `fload fstore`
(4) `aload` = `iconst fnewarray`
(5) `fnewarray` = `fnewarray astore aload`
(6) `iadd` = `fnewarray astore fnewarray astore iconst`
(7) `arraylength` = `astore iconst`
(8) `fadd` = `fstore fstore fload`
(9) `fconst` = `fload`
(10) `faload` = `fconst fastore fconst`

This set of equations is not unique, we could choose different ones. First part is obvious – we can easily check that equations between stack effects hold for a given set Δ . To prove that the languages coincide we do some more – we forget about Δ and examine the equations. If we get the same (actually isomorphic) Δ

proceeding from the equations then the goal is reached (Δ determines the equations and equations determine the same Δ).

Let us try to solve the following system in $\mathcal{S}$

(1) $(\ --\ ) = (s_1 -- s_2)\ (r_1 -- r_2)$
(2) $(\ --\ ) = (s_1 -- s_2)\ (m_1 -- m_2)\ (p_1 -- p_2)\ (x_1 -- x_2)$
(3) $(\ --\ ) = (u_1 -- u_2)\ (t_1 -- t_2)$
(4) $(s_1 -- s_2) = (m_1 -- m_2)\ (q_1 -- q_2)$
(5) $(q_1 -- q_2) = (q_1 -- q_2)\ (r_1 -- r_2)\ (s_1 -- s_2)$
(6) $(v_1 -- v_2) = (q_1 -- q_2)\ (r_1 -- r_2)\ (q_1 -- q_2)\ (r_1 -- r_2)\ (m_1 -- m_2)$
(7) $(z_1 -- z_2) = (r_1 -- r_2)\ (m_1 -- m_2)$
(8) $(w_1 -- w_2) = (t_1 -- t_2)\ (t_1 -- t_2)\ (u_1 -- u_2)$
(9) $(p_1 -- p_2) = (u_1 -- u_2)$
(10) $(y_1 -- y_2) = (p_1 -- p_2)\ (x_1 -- x_2)\ (p_1 -- p_2)$

First equation gives $s_1 = \Lambda$, $r_2 = \Lambda$ and $r_1 = s_2$, where Λ denotes the empty sequence. Let us define $\alpha = r_1\ (= s_2)$ for the future use. Now $s = (\ --\ \alpha)$ and $r = (\alpha --\ )$.

Fifth equation (together with facts we already know) allows to deduce that $\exists\beta \in \mathcal{T}^*$: $q_2 = \beta\alpha$.

Fourth equation can now be expressed as $(\ -- \alpha) = (m_1 -- m_2)\ (q_1 -- \beta\alpha)$ that gives us $m_1 = \Lambda$, $\exists\gamma \in \mathcal{T}^*$: $\alpha = \gamma\beta\alpha$ and $m_2 = \gamma q_1$. Consequently, $\gamma = \beta = \Lambda$ and we can define $\delta = m_2 = q_1$. Now $m = (\ --\ \delta)$ and $q = (\delta -- \alpha)$.

Equations (6) and (7) define $v = (\delta\ \delta\ --\ \delta)$ and $z = (\alpha\ --\ \delta)$.

Third equation is analogous to the (1): we have $u_1 = t_2 = \Lambda$ and we introduce a new sequence $\phi = u_2 = t_1$. Now $u = (\ --\ \phi)$ and $t = (\phi\ --\ )$.

Again, equations (8) and (9) allow direct calculations: $w = (\phi\ \phi\ --\ \phi)$ and $p = (\ --\ \phi)$.

Now it is time to use the second equation for $x = (smp)^{-1} = p^{-1}m^{-1}s^{-1}$ that gives us $x = (\alpha\ \delta\ \phi\ --\ )$.

Finally, from (10) we get $y = (\alpha\ \delta\ --\ \phi)$.

Let us sum up the result.

$m = sig(\texttt{iconst}) = (\ --\ \delta)$
$p = sig(\texttt{fconst}) = (\ --\ \phi)$
$q = sig(\texttt{fnewarray}) = (\delta\ --\ \alpha)$
$r = sig(\texttt{astore}) = (\alpha\ --\ )$
$s = sig(\texttt{aload}) = (\ --\ \alpha)$
$t = sig(\texttt{fstore}) = (\phi\ --\ )$
$u = sig(\texttt{fload}) = (\ --\ \phi)$
$v = sig(\texttt{iadd}) = (\delta\ \ \delta -- \delta)$
$w = sig(\texttt{fadd}) = (\phi\ \phi -- \phi)$
$x = sig(\texttt{fastore}) = (\alpha\ \delta\ \phi --)$
$y = sig(\texttt{faload}) = (\alpha\ \delta -- \phi)$
$z = sig(\texttt{arraylength}) = (\alpha -- \delta)$

We still have three independent variables $\alpha, \delta, \phi \in \mathcal{T}^*$ here and we have used all the equations. The result is isomorphic to the definitions in example 3.

As we have learned from this example the syntactic equations may determine stack effects as well as stack effects determine the syntax. At the same time it is not obvious how to choose such equations.

L i t e r a t u r e

1. [ClP67] Clifford A.H., Preston G.B. The algebraic theory of semigroups. Rhode Island, 1967.

2. [JVM95] The Java Virtual Machine Specification. Sun Microsystems, March 15, 1995, 74 pp.

3. [NiP70] Nivat M., Perrot J.F. Une généralisation du monoïde bicyclique. C.R.Acad.Sci. Paris, 271A, 1970, 824 – 827.

4. [Pöi90] Pöial J. Algebraic Specifications of Stack-effects for Forth Programs. 1990 FORML Conference Proceedings, EuroFORML'90 Conference, Oct 12 – 14, 1990, Ampfield, Nr Romsey, Hampshire, UK, Forth Interest Group, Inc., San Jose, USA, 1991, 282 – 290.

5. [Pöi91] Pöial J. Multiple Stack-effects of Forth Programs. 1991 FORML Conference Proceedings, euroFORML'91 Conference, Oct 11 – 13, 1991, Marianske Lazne, Czechoslovakia, Forth Interest Group, Inc., Oakland, USA, 1992, 400 – 406.

6. [Pöi94] Pöial J. Forth and Formal Language Theory. EuroForth'94, Nov 4 – 6, 1994, Winchester, UK, 1994, 47 – 52.

7. [StK93] Stoddart B., Knaggs P. Type Inference in Stack Based Languages. Formal Aspects of Computing, BCS, 1993, 5, 289 – 298.