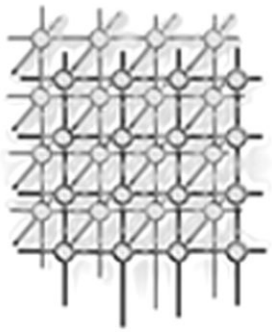
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:647–670 (DOI: 10.1002/cpe.798)

Simple verification technique for complex Java bytecode subroutines



Alessandro Coglio*,†

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

SUMMARY

Java is normally compiled to bytecode, which is verified and then executed by the Java Virtual Machine. Bytecode produced via compilation must pass verification. The main cause of complexity for bytecode verification is subroutines, used by compilers to generate more compact code. The techniques to verify subroutines proposed in the literature reject certain programs produced by mundane compilers, are difficult to realize within an implementation of the Java Virtual Machine or are relatively complicated. This paper presents a novel technique which is very simple to understand, implement and prove sound. It is also very powerful: the set of accepted programs has a simple characterization which most likely includes all the code produced by current compilers and which enables future compilers to make more extensive use of subroutines. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: Java; subroutines; bytecode verification

1. OVERVIEW

Java [1,2] is normally compiled to a platform-independent *bytecode* language, which is executed by the Java Virtual Machine (JVM) [3]. For security reasons [4,5] the JVM *verifies* incoming code prior to executing it, i.e. it checks certain type safety properties. Since compilers check equivalent properties on Java source, code produced via compilation must pass verification.

The main cause of complexity for bytecode verification is *subroutines*. Subroutines are internal to methods, invisible at the level of Java source. Compilers use them to generate more compact code [3, Section 7.13]. Without subroutines, the simple data flow analysis described in [3, Section 4.9.2] works beautifully well. With subroutines, in order to accept code produced by mundane compilers, a more precise analysis of the flow of control is needed.

*Correspondence to: Alessandro Coglio, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, U.S.A.
†E-mail: coglio@kestrel.edu

 Accepted 13 June 2003

In addition to the 'official' technique to verify subroutines informally described in [3, Section 4.9.6] and implemented in [6], several formal techniques have been proposed in the literature. Unfortunately, most of them (including the official one) reject certain programs produced by mundane compilers; the remaining techniques are difficult to realize within a JVM implementation or are relatively complicated. See Section 4 for details.

This paper presents a novel technique which is remarkably *simple* to understand, implement and prove sound. It is also extremely *powerful* in the sense that it accepts a quite large set of bytecode programs: accepted code has a very simple characterization which most likely includes all the code produced by current compilers and which enables future compilers to make a more extensive use of subroutines. While 'perfect' bytecode verification, like other forms of static analysis, is undecidable, the new technique can be argued to embody an optimal trade-off between power and simplicity.

The rest of this section summarizes the mathematical notations used in the paper. Section 2 describes subroutines and the key issues in their verification. Section 3 presents the new technique and its properties. Related work is discussed in Section 4. Appendix A collects the proofs of lemmas and theorems.

Mathematical notation

$\mathbf{N} = \{0, 1, 2, \dots\}$ is the set of natural numbers.

If A and B are sets, $A \to B$ is the set of all total functions from A to B. The notation $f : A \to B$ is equivalent to $f \in A \to B$. If f is a function, $\mathcal{D}(f)$ is the domain of f. The lambda notation $\lambda x.f(x)$ defines a function f, when $\mathcal{D}(f)$ is clear from the context. If $f : A \to B$, $a \in A$ and $b \in B$, $f\{a \mapsto b\}$ is the function $f' : A \to B$ defined by $f' = \lambda x.(\textbf{if } x = a \textbf{ then } b \textbf{ else } f(x))$, i.e. obtained by 'overwriting' the value of f at a to be b.

If $r \subseteq A \times A$ is a binary relation over a set A, $r^+ \subseteq A \times A$ is its transitive closure and $r^* \subseteq A \times A$ is its reflexive and transitive closure.

If A is a set, $\mathcal{P}_\omega(A)$ is the set of all finite subsets of A.

If A is a set, A^* is the set of all finite sequences of elements of A. If $s \in A^*$, $|s| \in \mathbf{N}$ is the length of s. A sequence $s \in A^*$ is also regarded as a function $s : \{i \in \mathbf{N} \mid i < |s|\} \to A$ (and vice versa) and $s(i)$ is written as s_i; note that the elements of s are numbered from 0 to $|s| - 1$, not from 1 to $|s|$. The notation $[s_0, \dots, s_{|s|-1}]$ displays all the elements of a sequence s in order. If $a \in A$ and $s \in A^*$, the notation $a \in s$ stands for $(\exists i \in \mathcal{D}(s).\ s_i = a)$. If $s \in A^*$ and $a \in A$, $s \cdot a$ is the sequence obtained by appending a to the right of s. $A^+ = \{s \in A^* \mid |s| \neq 0\}$ is the set of all non-empty sequences in A^*. If $s \in A^+$, $last(s)$ is the last (i.e. rightmost) element of s. If $n \in \mathbf{N}$, $A^{*(n)} = \{s \in A^* \mid |s| \leq n\}$ is the set of all sequences in A^* of length at most n.

If A and B are sets, $A_\mathrm{T} \uplus B_\mathrm{U}$ is the disjoint union of A and B using subscripts T and U to tag the elements from A and B. So, if $a \in A$ and $b \in B$ then $a_\mathrm{T}, b_\mathrm{U} \in A_\mathrm{T} \uplus B_\mathrm{U}$. This generalizes to three or more sets.

A lattice is a quadruple $\langle L, \sqsubseteq, \sqcap, \sqcup \rangle$ where: L is a set; $\sqsubseteq \subseteq L \times L$ is a partial order relation over L, i.e. it is reflexive ($x \sqsubseteq x$), anti-symmetric (if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$) and transitive (if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$); $\sqcap : L \times L \to L$ is a binary operation over L, called *meet*, which returns the greatest lower bound of its arguments ($x \sqcap y \sqsubseteq x$, $x \sqcap y \sqsubseteq y$ and if $z \sqsubseteq x$ and $z \sqsubseteq y$, then $z \sqsubseteq x \sqcap y$); and $\sqcup : L \times L \to L$ is a binary operation over L, called *join*, which returns the least upper bound of its arguments ($x \sqcup y \sqsupseteq x$, $x \sqcup y \sqsupseteq y$ and if $z \sqsupseteq x$ and $z \sqsupseteq y$, then $z \sqsupseteq x \sqcup y$). Both $\sqcap$ and $\sqcup$ can be shown

to be commutative ($x \sqcap y = y \sqcap x$), idempotent ($x \sqcap x = x$) and associative ($(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$). If L is finite, it contains a *top* element $\top$ such that $\top \sqsupseteq x$ for all $x \in L$, as well as a *bottom* element $\bot$ such that $\bot \sqsubseteq x$ for all $x \in L$.

2. SUBROUTINES

This section defines the syntax and semantics of a simple language $\mathcal{L}$ with subroutines, similar to the languages used in [7–11], along with a notion of type safety. The key issues in verification are then discussed. $\mathcal{L}$ is an abstraction of Java bytecode, which is much richer. This simpler language exposes the essence of problems and solutions, because the omitted features are orthogonal.

2.1. A simple language with subroutines

The *instructions* of $\mathcal{L}$ are drawn from

$$\begin{aligned} \mathit{Instr} = {} & \{\mathsf{halt}, \mathsf{push0}, \mathsf{inc}, \mathsf{div}, \mathsf{pop}\} \cup \{\mathsf{if0}\ j \mid j \in \mathbf{N}\} \\ & \cup \{\mathsf{load}\ x \mid x \in \mathit{VN}\} \cup \{\mathsf{store}\ x \mid x \in \mathit{VN}\} \\ & \cup \{\mathsf{jsr}\ s \mid s \in \mathbf{N}\} \cup \{\mathsf{ret}\ x \mid x \in \mathit{VN}\} \end{aligned}$$

where VN is a finite set of *variable names* whose exact definition is immaterial.

A *program* in $\mathcal{L}$ is a finite, non-empty sequence of instructions,

$$P \in \mathit{Instr}^+$$

satisfying the following requirements:

1. $(\mathsf{if0}\ j) \in P \Rightarrow j \in \mathcal{D}(P)$;
2. $(\mathsf{jsr}\ s) \in P \Rightarrow s \in \mathcal{D}(P)$;
3. $\mathit{last}(P) = \mathsf{halt} \vee (\exists x.\ \mathit{last}(P) = \mathsf{ret}\ x)$;
4. $(\exists x.(\mathsf{ret}\ x) \in P) \Rightarrow (\exists s.\ (\mathsf{jsr}\ s) \in P)$.

The first three requirements constrain the flow of control never to go outside P during execution (see below)[‡]. The fourth requirement, which is reasonable, slightly simplifies verification, but it can easily be removed [12] if desired. The elements of $\mathcal{D}(P)$ are called the *addresses* of P. The addresses in

$$S = \{s \in \mathcal{D}(P) \mid (\mathsf{jsr}\ s) \in P\} \quad \text{and} \quad C = \{c \in \mathcal{D}(P) \mid \exists s.P_c = \mathsf{jsr}\ s\}$$

are called *subroutine addresses* and *calling addresses*[§].

The *values* on which P operates are drawn from

$$\mathit{Val} = \mathit{Int}_{\mathrm{I}} \uplus \mathit{Flt}_{\mathrm{F}} \uplus C_{\mathrm{C}}$$

[‡]While in the JVM these requirements are checked by the bytecode verifier, they have been incorporated into the definition of programs because they are straightforward to check.

[§]The notation for S and C does not reflect their dependence on P, which is left implicit for readability. This also applies to other mathematical entities introduced below.

where *Int* and *Flt* are sets of *integers* and *floats* whose precise definition is immaterial, as long as the following requirements are satisfied:

1. $0 \in Int$;
2. $\iota \in Int \Rightarrow inc(\iota) \in Int$;
3. $\iota, \iota' \in Int \Rightarrow div(\iota, \iota') \in Flt$.

The functions *inc* and *div* capture increment by 1 and division; for simplicity, *div* yields a result (e.g. 0) even if the divisor is 0.

Values are stored in two areas: *variables* and *stack*. While the latter is accessed in a last-in, first-out fashion, the former are directly accessed via the names in *VN*. Contents of the variables and stack are thus captured by

$$Var = VN \to Val \quad \text{and} \quad Stk = Val^{*(max)}$$

where *max* is a limit to the size of the stack whose exact definition is immaterial.

States of execution are elements of

$$Stt = (\mathcal{D}(P) \times Var \times Stk) \cup \{\mathsf{Err}\}$$

Err is the *error state*, which arises when some type-unsafe operation is attempted (see below). The first component of a state of the form $\langle i, vr, sk \rangle$ is the *program counter*, i.e. the address of the instruction about to be executed; the other two components are the current contents of the variables and stack. The *initial state* is

$$\mathsf{Init} = \langle 0, \lambda x.0_{\mathrm{I}}, [\,] \rangle \in Stt$$

Execution is formalized by a *transition* relation

$$\rightsquigarrow \subseteq Stt \times Stt$$

defined as the smallest satisfying the last two rules in Figure 1, which reference another relation $\rightsquigarrow_0 \subseteq Stt \times Stt$, defined as the smallest satisfying the other rules in Figure 1. Each of the rules defining $\rightsquigarrow_0$ describes the execution of an instruction; each instruction execution is a transition, as expressed by (OK). There is no rule for halt: the program terminates when halt is reached. Each of the rules defining $\rightsquigarrow_0$ includes type safety checks (some in the form of patterns at the left of the $\rightsquigarrow_0$ symbol) ensuring that no stack overflow or underflow occurs and that no operation is performed on values of the wrong type (e.g. it is not possible to 'increment a float'). If such checks are not satisfied, (ER) is applicable: its second condition requires that none of the rules defining $\rightsquigarrow_0$ be applicable, which happens exactly when some type safety condition for the instruction at the program counter is not satisfied. The application of (ER) causes the state to become Err, from which no further transition can take place.

Typical implementations of the JVM do not perform these type safety checks for performance reasons. It is the task of bytecode verification to statically ensure that these checks would succeed if they were performed at run-time. The outcome of performing an operation on values of the wrong types is undefined [3, Section 6.1]; Err is an abstraction of the undefined state into which a JVM implementation would move. The following formal notion of type safety is thus introduced:

$$TypeSafe(P) \Leftrightarrow \mathsf{Init} \not\rightsquigarrow^{+} \mathsf{Err}$$

$$\frac{P_i = \mathsf{push0} \quad |sk| < max}{\langle i, vr, sk\rangle \ \leadsto_0 \ \langle i+1, vr, sk \cdot 0_{\mathrm{I}}\rangle} \ (\mathrm{PH})$$

$$\frac{P_i = \mathsf{inc}}{\langle i, vr, sk \cdot \iota_{\mathrm{I}}\rangle \ \leadsto_0 \ \langle i+1, vr, sk \cdot inc(\iota)_{\mathrm{I}}\rangle} \ (\mathrm{IN})$$

$$\frac{P_i = \mathsf{div}}{\langle i, vr, sk \cdot \iota_{\mathrm{I}} \cdot \iota'_{\mathrm{I}}\rangle \ \leadsto_0 \ \langle i+1, vr, sk \cdot div(\iota', \iota)_{\mathrm{F}}\rangle} \ (\mathrm{DV})$$

$$\frac{P_i = \mathsf{pop}}{\langle i, vr, sk \cdot v\rangle \ \leadsto_0 \ \langle i+1, vr, sk\rangle} \ (\mathrm{PP})$$

$$\frac{P_i = \mathsf{load}\ x \quad |sk| < max}{\langle i, vr, sk\rangle \ \leadsto_0 \ \langle i+1, vr, sk \cdot vr(x)\rangle} \ (\mathrm{LD})$$

$$\frac{P_i = \mathsf{store}\ x}{\langle i, vr, sk \cdot v\rangle \ \leadsto_0 \ \langle i+1, vr\{x \mapsto v\}, sk\rangle} \ (\mathrm{ST})$$

$$\frac{P_i = \mathsf{if0}\ j}{\langle i, vr, sk \cdot \iota_{\mathrm{I}}\rangle \ \leadsto_0 \ \langle (\mathbf{if}\ \iota = 0\ \mathbf{then}\ j\ \mathbf{else}\ i+1), vr, sk\rangle} \ (\mathrm{IF})$$

$$\frac{P_i = \mathsf{jsr}\ s \quad |sk| < max}{\langle i, vr, sk\rangle \ \leadsto_0 \ \langle s, vr, sk \cdot i_{\mathrm{C}}\rangle} \ (\mathrm{JS})$$

$$\frac{P_i = \mathsf{ret}\ x \quad vr(x) = c_{\mathrm{C}}}{\langle i, vr, sk\rangle \ \leadsto_0 \ \langle c+1, vr, sk\rangle} \ (\mathrm{RT})$$

$$\frac{stt \ \leadsto_0 \ stt'}{stt \ \leadsto \ stt'} \ (\mathrm{OK})$$

$$\frac{P_i \neq \mathsf{halt} \quad \nexists\, i', vr', sk'.\ \langle i, vr, sk\rangle \ \leadsto_0 \ \langle i', vr', sk'\rangle}{\langle i, vr, sk\rangle \ \leadsto \ \mathsf{Err}} \ (\mathrm{ER})$$

Figure 1. Rules defining the operational semantics of $\mathcal{L}$.

The jsr and ret instructions realize subroutines by saving addresses and later returning to them. However, there is no explicit notion of subroutine as a textually delimited piece of code: jsr and ret may be scattered here and there in P. While compilers usually produce code where the address range of each subroutine is clearly identifiable, certain Java programs result in bytecode where subroutines

can be exited implicitly via branching or exceptions (see Section 4), making the determination of their boundaries more difficult.

2.2. Requirements for verification

The purpose of verification is to establish whether a program in $\mathcal{L}$ is type-safe. Verification must be *sound*: if a program is accepted then it is definitely type-safe. Due to the usual undecidability problems, verification is bound to be *incomplete*: some type-safe programs are unjustly rejected. Anyhow, verification must accept at least all the programs in $\mathcal{L}$ that are abstractions of bytecode produced by Java compilers.

The last requirement is not as easy to assess because there is currently no precise characterization of the output of Java compilers, which is furthermore susceptible to change as compilers evolve. As advocated in [13], a solution is to use a precise characterization of a set of bytecode programs as a 'contract' between compiler developers and JVM developers: the former shall write compilers whose produced programs belong to the set and the latter shall write bytecode verifiers that accept any program belonging to the set. The current lack of such a contract may be related to the fact that existing bytecode verifiers reject certain programs produced by existing compilers (see Section 4).

2.3. Complexity caused by subroutines

In [3, Section 4.9.2] a technique for bytecode verification is informally described which is a (forward) data flow analysis [14]. This technique is implemented in [6] and formalized in [15]. The analysis assigns to each address $i \in \mathcal{D}(P)$ type information vt_i and st_i for the variables and stack: vt_i is a function from VN to types and st_i is a sequence of types of length at most max. Types include int for integers, flt for floats and any for all values.

The assignment is computed iteratively. Types for Init are assigned to address 0 and then propagated through all possible control paths in the program, transforming them according to the instructions encountered along the paths (e.g. if $st_i = st \cdot \mathsf{int} \cdot \mathsf{int}$ and $P_i = \mathsf{div}$ then $st_{i+1} = st \cdot \mathsf{flt}$). Types from converging paths are merged point-wise on the variables and stack elements; the result of merging different types (e.g. int and flt) is any. If the types at address i do not match the requirements of the instruction P_i (e.g. $st_i = [\,]$ and $P_i = \mathsf{pop}$), then verification fails because the program may be type-unsafe. Otherwise, a fixed point is eventually reached with a consistent type assignment that witnesses the type safety of the program; an example is shown in Figure 2 (it is assumed that $max \geq 2$).

Consider now the program in Figure 3, where the subroutine at addresses 11–12 saves the calling address (4 or 7) into y and then immediately returns to its successor (5 or 8). When the subroutine is called from 4, x contains a float; when called from 7, an integer. This means that address 11 is assigned type any for variable x: this is propagated, through the ret, to addresses 5 and 8. So, inc causes verification to fail because the top of the stack is not int. The type ca is used for calling addresses[¶].

[¶]This is a simplification, because types for calling addresses must be qualified by subroutine addresses (i.e. ca_{11}) in order to distinguish between different subroutines [6, 7, 10, 11, 15–17]. However, this is irrelevant to the example in Figure 3, where there is only one subroutine.

i	P_i	st_i	$vt_i(x)$
0	push0	[ ]	int
1	pop	[int]	int
2	push0	[ ]	any
3	push0	[int]	any
4	div	[int, int]	any
5	store x	[flt]	any
6	push0	[ ]	flt
7	if0 2	[int]	flt
8	halt	[ ]	flt

Figure 2. Successful verification of a program in $\mathcal{L}$ without subroutines.

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$
0	push0	[ ]	int	int
1	push0	[int]	int	int
2	div	[int, int]	int	int
3	store x	[flt]	int	int
4	jsr 11	[ ]	flt	int
5	push0	[ ]	any	ca
6	store x	[int]	any	ca
7	jsr 11	[ ]	int	ca
8	load x	[ ]	any	ca
9	inc	[any]	any	ca
10	halt	fail		
11	store y	[ca]	any	any
12	ret y	[ ]	any	ca

Figure 3. Unsuccessful verification of a program in $\mathcal{L}$ with subroutines.

Despite the failure of verification, the program in Figure 3 is type-safe. The mismatch is due to the impossibility at run-time to call the subroutine from 4 and then return to 8, or call it from 7 and then return to 5. If called from 4, control can only return to 5; if from 7, only to 8. However, the analysis does not discern these possible and impossible paths. While inside the subroutine any is accurate (because it is called from two addresses with incompatible types for x), the types at the calling addresses should be 'retained' at the successors of the calling addresses. In other words, a more precise analysis of the flow of control of the subroutine is needed, in order to determine more accurate types for the program.

Unfortunately, compilers produce code similar to Figure 3, where a variable has different types inside a subroutine [3, Section 4.9.6]. The solution prescribed in [3, Section 4.9.6] and implemented in [6] is to

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$
0	push0	[ ]	int	int
1	push0	[int]	int	int
2	div	[int, int]	int	int
3	store x	[flt]	int	int
4	jsr 11	[ ]	flt	int
5	push0	[ ]	flt	ca_4
6	store x	[int]	flt	ca_4
7	jsr 11	[ ]	int	ca_4
8	load x	[ ]	int	ca_7
9	inc	[int]	int	ca_7
10	halt	[int]	int	ca_7
11	store y	[ca_4 | ca_7]	flt | int	int | ca_4
12	ret y	[ ]	flt | int	ca_4 | ca_7

Figure 4. Successful verification of the program in Figure 3.

keep track of which variables are modified inside a subroutine: if a variable is not marked as modified, its type at $c+1$ is propagated from c and not from the address of the ret. Other verification techniques [7,11,16,17] use a similar approach. A thorough study of this solution, along with its problems and ways to fix some of them, is given in [15]: the bottom line is that while it works for simpler programs such as the one in Figure 3, it rejects less simple programs that are nonetheless produced by mundane compilers (see Section 4).

3. THE TECHNIQUE

This section first provides the intuition behind the new technique and the reason why it works. Then it formally defines it and presents its properties in the form of theorems. Finally, some implementation issues are discussed.

3.1. Intuition

Consider again the program in Figure 3, copied in Figure 4. The reason why verification fails is that the types int and flt for x assigned to calling addresses 4 and 7 are irreversibly merged into type any at subroutine address 11. From any alone at address 12 there is no way to restore int and flt at the successors 5 and 8 of the calling addresses. This consideration prompts the first key idea: instead of merging int and flt into any, both are kept. The meaning of flt | int is that the type is either flt or int and opens the possibility of separating them when returning from the subroutine.

How can the verifier decide to propagate flt to address 5 and int to address 8? The second key idea is to qualify ca with calling addresses. So, by uniformly keeping all type alternatives for every variable

and stack element, address 11 is assigned $\mathsf{ca}_4 \mid \mathsf{ca}_7$ for the stack. This is moved into variable y and now the ret instruction can propagate the types on the left of the | symbol (flt for x) to address 5 and the types on the right (int for x) to address 8, because ca_4 is on the left and ca_7 on the right. Since the stack contains int at address 9, inc does not cause any failure and the program is happily accepted.

This very simple approach scales from the example in Figure 4 to arbitrarily complex programs. Consider a program with several subroutine calls and returns. During verification, at some addresses of the program there will be several type alternatives, including types of the form ca_c. At any ret x instruction the type alternatives $\mathsf{ca}_{c_1} \mid \ldots \mid \mathsf{ca}_{c_n}$ for x are used to select which types must be propagated to each address $c_i + 1$. The other types can be safely left out, because they correspond to impossible paths at run-time.

3.2. Definition

The technique fits into the data flow analysis framework [14]. It is defined by: a lattice whose elements capture the type information assigned to addresses and whose join operation captures the merging of type information from converging paths in the program; transfer functions capturing the transformation of type information by instructions; and data flow constraints whose solution determines whether a program passes verification or not.

The notation of type alternatives separated by the | symbol is useful to help intuition, but it would not work in a lattice because the merging operation $\sqcup$ must be commutative and idempotent, while the order and number of the type alternatives is critical. So, the idea is to use finite sets whose elements are complete type assignments to the variables and stack elements.

The starting point is the set

$$Type = \{\mathsf{int}, \mathsf{flt}\} \cup \{\mathsf{ca}_c \mid c \in C\}$$

of *types*. Type assignments to the variables and stack are captured by

$$VType = VN \to Type \quad \text{and} \quad SType = Type^{*(max)}$$

Abbreviating $VSType = VType \times SType$, the lattice

$$\langle L, \sqsubseteq, \sqcap, \sqcup \rangle$$

is defined by:

1. $L = \mathcal{P}_\omega(VSType) \cup \{\mathsf{fail}\}$;
2. $l \in L \Rightarrow l \sqsubseteq \mathsf{fail}$;
3. $l, l' \in \mathcal{P}_\omega(VSType) \Rightarrow (l \sqsubseteq l' \Leftrightarrow l \subseteq l')$.

In other words, the lattice $\langle \mathcal{P}_\omega(VSType), \subseteq, \cap, \cup \rangle$ is augmented with an extra top element fail. Since $Type$, VN and max are finite, L is also finite. As shown later, the meaning of a lattice element $l \in \mathcal{P}_\omega(VSType)$ assigned to address i is that at run-time, whenever the program counter is i, the values in the variables and stack have the types of some pair $\langle vt, st \rangle \in l$. The join operation $\sqcup$ is very simple: the result of merging two sets of pairs is their union, while merging fail with anything yields fail.

The transfer functions $tf : L \to L$ are defined in Figure 5. With the exception of the one for ret, they all operate element-wise on each set of $\mathcal{P}_\omega(VSType)$. They capture the effect of the corresponding

$$
\begin{aligned}
tf_{\mathsf{push0}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ |st| < \mathit{max}\ \textbf{then}\ \langle vt, st \cdot \mathsf{int}\rangle\ \textbf{else}\ \mathsf{fail})\\
tf_{\mathsf{inc}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ st = st' \cdot \mathsf{int}\ \textbf{then}\ \langle vt, st\rangle\ \textbf{else}\ \mathsf{fail})\\
tf_{\mathsf{div}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ st = st' \cdot \mathsf{int} \cdot \mathsf{int}\ \textbf{then}\ \langle vt, st' \cdot \mathsf{flt}\rangle\ \textbf{else}\ \mathsf{fail})\\
tf_{\mathsf{pop}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ st = st' \cdot t\ \textbf{then}\ \langle vt, st'\rangle\ \textbf{else}\ \mathsf{fail})\\
tf^{x}_{\mathsf{load}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ |st| < \mathit{max}\ \textbf{then}\ \langle vt, st \cdot vt(x)\rangle\ \textbf{else}\ \mathsf{fail})\\
tf^{x}_{\mathsf{store}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ st = st' \cdot t\ \textbf{then}\ \langle vt\{x \mapsto t\}, st'\rangle\ \textbf{else}\ \mathsf{fail})\\
tf_{\mathsf{if0}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ st = st' \cdot \mathsf{int}\ \textbf{then}\ \langle vt, st'\rangle\ \textbf{else}\ \mathsf{fail})\\
tf^{c}_{\mathsf{jsr}} &= \mathit{lift}(\lambda\langle vt, st\rangle.\ \textbf{if}\ |st| < \mathit{max}\ \textbf{then}\ \langle vt, st \cdot \mathsf{ca}_c\rangle\ \textbf{else}\ \mathsf{fail})\\
tf^{x,c}_{\mathsf{ret}}(l) &= \textbf{if}\ (l \neq \mathsf{fail} \wedge (\forall\langle vt, st\rangle \in l.(\exists c'.vt(x) = \mathsf{ca}_{c'})))\\
&\qquad \textbf{then}\ \{\langle vt, st\rangle \in l \mid vt(x) = \mathsf{ca}_c\}\ \textbf{else}\ \mathsf{fail}
\end{aligned}
$$

Figure 5. Transfer functions for $\mathcal{L}$.

instructions on the types for the variables and stack, and can be derived from the rules in Figure 1. If any type safety condition of those rules is not satisfied, the transfer function returns fail, which is propagated by all transfer functions. So, it is convenient to define all these transfer functions by means of the higher-order function

$$\mathit{lift} : (\mathit{VSType} \to \mathit{VSType} \cup \{\mathsf{fail}\}) \to (L \to L)$$

defined by

$$
\begin{aligned}
\mathit{lift}(f)(l) = \textbf{if}\ &(l \neq \mathsf{fail} \wedge (\forall\langle vt, st\rangle \in l.f(vt, st) \neq \mathsf{fail}))\\
&\textbf{then}\ \{f(vt, st) \mid \langle vt, st\rangle \in l\}\ \textbf{else}\ \mathsf{fail}
\end{aligned}
$$

The transfer functions for load and store are parameterized by the variable name x that is part of the instruction. The transfer function for jsr is parameterized by the calling address at which the jsr appears in P.

The transfer function for ret operates on the sets by filtering them with respect to a certain calling address c: only the elements that have type ca_c assigned to variable x are kept, while the others are discarded. The transfer function is parameterized by the variable name that is part of the instruction and by a calling address (see below).

All the transfer functions in Figure 5 are monotone, i.e.

$$l \sqsubseteq l' \Rightarrow tf(l) \sqsubseteq tf(l')$$

This is easily proved because larger sets are mapped to larger sets or to fail, which is the top of the lattice, and fail is mapped to fail.

The set of constraints derived from P is

$$\mathit{Constr} = \{u_0 \sqsupseteq l_{\mathsf{Init}}\} \cup \bigcup_{i \in \mathcal{D}(P)} \mathit{IConstr}(i, P_i)$$

$$
\begin{aligned}
&\mathit{IConstr}(i, \mathsf{push0}) = \{u_{i+1} \sqsupseteq \mathit{tf}_{\mathsf{push0}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{inc}) = \{u_{i+1} \sqsupseteq \mathit{tf}_{\mathsf{inc}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{div}) = \{u_{i+1} \sqsupseteq \mathit{tf}_{\mathsf{div}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{pop}) = \{u_{i+1} \sqsupseteq \mathit{tf}_{\mathsf{pop}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{load}\ x) = \{u_{i+1} \sqsupseteq \mathit{tf}^{x}_{\mathsf{load}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{store}\ x) = \{u_{i+1} \sqsupseteq \mathit{tf}^{x}_{\mathsf{store}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{if0}\ j) = \{u_{i+1} \sqsupseteq \mathit{tf}_{\mathsf{if0}}(u_i), u_j \sqsupseteq \mathit{tf}_{\mathsf{if0}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{jsr}\ s) = \{u_s \sqsupseteq \mathit{tf}^{i}_{\mathsf{jsr}}(u_i)\}\\
&\mathit{IConstr}(i, \mathsf{ret}\ x) = \{u_{c+1} \sqsupseteq \mathit{tf}^{x,c}_{\mathsf{ret}}(u_i) \mid c \in C\}\\
&\mathit{IConstr}(i, \mathsf{halt}) = \emptyset
\end{aligned}
$$

Figure 6. Constraints derived from the instructions of $\mathcal{L}$.

where

$$l_{\mathsf{Init}} = \{\langle \lambda x.\mathsf{int}, [\,]\rangle\}$$

is the lattice element that captures types for Init and the constraints derived from each instruction are defined in Figure 6.

Each u_i is a placeholder for a lattice element assigned to address i. Each constraint has the form $u_{i'} \sqsupseteq \mathit{tf}(u_i)$, except for the initial constraint $u_0 \sqsupseteq l_{\mathsf{Init}}$. The former expresses that the types at address i' must 'include' (i.e. $\sqsupseteq$) the result of transforming the types at address i according to the transfer function tf for instruction P_i. The latter expresses that the types at address 0 must include the types for Init. A solution to *Constr* is an assignment that satisfies all the constraints.

Note that a ret instruction contributes one constraint for each calling address in P; the calling address is used as the second parameter of the transfer function. If no pair of l contains ca_c in variable x, then $\mathit{tf}^{x,c}_{\mathsf{ret}}(l) = \emptyset$, i.e. the constraint $u_{c+1} \sqsupseteq \mathit{tf}^{x,c}_{\mathsf{ret}}(u_i)$ effectively contributes no types to address $c+1$ (because $\emptyset$ is the bottom of the lattice), reflecting the fact that control cannot be transferred from i to $c+1$.

Since L is finite and all the transfer functions are monotone, the set of constraints has always a least solution $\sigma : \mathcal{D}(P) \to L$, which can be efficiently computed iteratively [14,15]. The notion of verification is defined as

$$\mathit{Verified}(P) \Leftrightarrow (\forall i \in \mathcal{D}(P).\sigma_i \neq \mathsf{fail})$$

As an example, the constraints derived from the program in Figure 4 are shown in Figure 7. The least solution to those constraints is shown in Figure 8 in terms of explicit sets of pairs (while Figure 4 shows the solution in terms of type alternatives).

The technique is very simple to understand and implement. Unlike others, it does not attempt to determine subroutine boundaries or variables modified inside subroutines. Rather, its 'unstructured'

$$
\begin{aligned}
u_0 &\sqsupseteq l_{\mathsf{Init}} \\
u_1 &\sqsupseteq tf_{\mathsf{push0}}(u_0) \\
u_2 &\sqsupseteq tf_{\mathsf{push0}}(u_1) \\
u_3 &\sqsupseteq tf_{\mathsf{div}}(u_2) \\
u_4 &\sqsupseteq tf^{x}_{\mathsf{store}}(u_3) \\
u_{11} &\sqsupseteq tf^{4}_{\mathsf{jsr}}(u_4) \\
u_6 &\sqsupseteq tf_{\mathsf{push0}}(u_5) \\
u_7 &\sqsupseteq tf^{x}_{\mathsf{store}}(u_6) \\
u_{11} &\sqsupseteq tf^{7}_{\mathsf{jsr}}(u_7) \\
u_9 &\sqsupseteq tf^{x}_{\mathsf{load}}(u_8) \\
u_{10} &\sqsupseteq tf_{\mathsf{inc}}(u_9) \\
u_{12} &\sqsupseteq tf^{y}_{\mathsf{store}}(u_{11}) \\
u_5 &\sqsupseteq tf^{y,4}_{\mathsf{ret}}(u_{12}) \\
u_8 &\sqsupseteq tf^{y,7}_{\mathsf{ret}}(u_{12})
\end{aligned}
$$

Figure 7. Constraints derived from the program in Figure 4.

nature reflects the possibly unstructured occurrences of jsr and ret in programs. Its treatment of jsr and ret is as simple as their run-time behavior, described by the rules (JS) and (RT).

3.3. Soundness

It is quite simple to prove that the technique is sound. The key invariant at run-time is that when the program counter is i there is a pair $\langle vt, st\rangle \in \sigma_i$ containing the current types of the variables and stack. So, if the program attempted a type-unsafe operation at this point, $tf(\sigma_i) = \mathsf{fail}$ (where tf is the transfer function associated to P_i) would appear at some address i' of the program, because of the constraint $\sigma_{i'} \sqsupseteq tf(\sigma_i)$; but if P is verified, that cannot happen.

The invariant is trivially true at Init and is preserved by every execution step because every transfer function maps each pair $\langle vt, st\rangle \in \sigma_i$ to a pair $\langle vt', st'\rangle \in \sigma_{i'}$ transforming the types according to the instruction; this mapping includes the pair containing the current types for the variables and stack. In the case of $tf^{x,c}_{\mathsf{ret}}$, not all pairs are mapped, but those that are discarded do not contain ca_c for variable x and so none of the discarded pairs can be the one containing the current types of the variables and stack.

$$\sigma_0 = \{\langle \lambda z.\mathsf{int}, [\,]\rangle\}$$
$$\sigma_1 = \{\langle \lambda z.\mathsf{int}, [\mathsf{int}]\rangle\}$$
$$\sigma_2 = \{\langle \lambda z.\mathsf{int}, [\mathsf{int}, \mathsf{int}]\rangle\}$$
$$\sigma_3 = \{\langle \lambda z.\mathsf{int}, [\mathsf{flt}]\rangle\}$$
$$\sigma_4 = \{\langle \lambda z.\mathsf{int}\{x \mapsto \mathsf{flt}\}, [\,]\rangle\}$$
$$\sigma_5 = \{\langle \lambda z.\mathsf{int}\{x \mapsto \mathsf{flt}\}\{y \mapsto \mathsf{ca}_4\}, [\,]\rangle\}$$
$$\sigma_6 = \{\langle \lambda z.\mathsf{int}\{x \mapsto \mathsf{flt}\}\{y \mapsto \mathsf{ca}_4\}, [\mathsf{int}]\rangle\}$$
$$\sigma_7 = \{\langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_4\}, [\,]\rangle\}$$
$$\sigma_8 = \{\langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_7\}, [\,]\rangle\}$$
$$\sigma_9 = \{\langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_7\}, [\mathsf{int}]\rangle\}$$
$$\sigma_{10} = \{\langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_7\}, [\mathsf{int}]\rangle\}$$
$$\sigma_{11} = \{\langle \lambda z.\mathsf{int}\{x \mapsto \mathsf{flt}\}, [\mathsf{ca}_4]\rangle, \langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_4\}, [\mathsf{ca}_7]\rangle\}$$
$$\sigma_{12} = \{\langle \lambda z.\mathsf{int}\{x \mapsto \mathsf{flt}\}\{y \mapsto \mathsf{ca}_4\}, [\,]\rangle, \langle \lambda z.\mathsf{int}\{y \mapsto \mathsf{ca}_7\}, [\,]\rangle\}$$

Figure 8. Least solution to the constraints in Figure 7.

Types of values are captured by the *abstraction* function

$$\alpha : \mathit{Val} \to \mathit{Type} \quad \text{defined by} \quad \alpha(\iota_1) = \mathsf{int} \wedge \alpha(\phi_{\mathrm{F}}) = \mathsf{flt} \wedge \alpha(c_{\mathrm{C}}) = \mathsf{ca}_c$$

The abstraction function is lifted to the variables and stack point-wise:

$$\alpha(vr) = \lambda x.\, \alpha(vr(x)) \quad \text{and} \quad \alpha(sk) = [\alpha(sk_0), \ldots, \alpha(sk_{|sk|-1})]$$

So, the invariant is formalized as

$$\mathit{Inv}(stt) \Leftrightarrow \exists i, vr, sk.(stt = \langle i, vr, sk\rangle \wedge \langle \alpha(vr), \alpha(sk)\rangle \in \sigma_i)$$

Note that Err does not satisfy the invariant.

Lemma 1. $\mathit{Verified}(P) \Rightarrow \mathit{Inv}(\mathsf{Init})$.

Lemma 2. $\mathit{Verified}(P) \wedge \mathit{Inv}(stt) \wedge stt \rightsquigarrow stt' \Rightarrow \mathit{Inv}(stt')$.

Theorem 1. (Soundness) $\mathit{Verified}(P) \Rightarrow \mathit{TypeSafe}(P)$.

3.4. Characterization

The example in Figure 9 shows that the technique is incomplete. Since the technique is insensitive to the actual integer value in the stack at address 1 (it is just an int), it cannot recognize that address 2 is unreachable.

i	P_i	st_i
0	push0	[ ]
1	if0 3	[int]
2	div	[ ]
3	halt	fail

Figure 9. Unsuccessful verification of a type-safe program.

$$\frac{stt \rightsquigarrow stt'}{stt \rightsquigarrow_{\mathrm{I}} stt'}\ (\mathrm{EX})$$

$$\frac{P_i = \mathsf{if0}\ j}{\langle i, vr, sk \cdot \iota_{\mathrm{I}}\rangle \rightsquigarrow_{\mathrm{I}} \langle(\mathbf{if}\ \iota \neq 0\ \mathbf{then}\ j\ \mathbf{else}\ i+1), vr, sk\rangle}\ (\mathrm{IF'})$$

Figure 10. Rules defining the integer-insensitive operational semantics of $\mathcal{L}$.

The if0 instruction is the only cause of incompleteness because all integers are approximated by type int, as opposed to calling addresses, which are isomorphic to their types. This suggests that, if if0 were insensitive to the actual value of the integer at the top of the stack, the technique would be complete. So, consider an 'integer-insensitive' operational semantics of $\mathcal{L}$

$$\rightsquigarrow_{\mathrm{I}} \subseteq Stt \times Stt$$

defined as the smallest relation satisfying the rules in Figure 10. The rule (EX) says that $\rightsquigarrow_{\mathrm{I}}$ is an extension of $\rightsquigarrow$. The rule (IF') is exactly like (IF) but with the test negated. Jointly, (IF) and (IF') allow the execution of an if0 instruction to non-deterministically transfer control to either the successor address or the target of the if0, regardless of the value of the integer. The notion of integer-insensitive type safety is defined in complete analogy with Section 2.1:

$$TypeSafe_{\mathrm{I}}(P) \Leftrightarrow \mathsf{Init} \not\rightsquigarrow_{\mathrm{I}}{}^{+} \mathsf{Err}$$

Theorem 2. (Characterization) $Verified(P) \Leftrightarrow TypeSafe_{\mathrm{I}}(P)$.

The theorem provides a very simple and precise characterization of which programs are accepted by verification: exactly all type-safe programs in the integer-insensitive operational semantics. To date, the author has not found any bytecode program generated by a compiler that does not satisfy this characterization[‖]. Since compilers are quite unlikely to expect bytecode verifiers to be integer-sensitive, there are reasons to believe that the characterization includes all the code produced by current and

[‖]For Java bytecode, the notion of insensitivity must be extended from integers to null references and other features [12]. For simplicity, the restrictive term 'integer-insensitive' and related ones are also used in the rest of this paper when referring to Java bytecode.

future compilers. Anyhow, this conjecture should be further validated by experimentation (i.e. running compilers and checking the generated code) and formal studies (e.g. formalizing compilation strategies, as in [17], and proving that the produced code satisfies the characterization).

The characterization includes programs that are not generated by current compilers: for instance, programs with polymorphic subroutines to swap the values of two variables, or to make duplicates of the top value of the stack, even with stacks of different sizes**. This enables future compilers to make more extensive use of subroutines in order to generate more compact code. However, it is presently unclear how much additional space could be actually saved in code produced from mundane Java programs.

It is certainly possible to devise verification techniques that accept more type-safe programs than the new technique. For example, the type int for integers could be refined into a type for 0_I and a type for possibly non-zero integers. The transfer function for if0 could filter away from the successor address all the pairs $\langle vt, st \rangle$ that have the type for 0_I at the top of the stack. The program in Figure 9 would then be accepted. However, this refinement makes verification more complicated and invalidates the simple characterization provided by Theorem 2. In addition, the benefit is dubious: no sensible compiler would ever generate a program like the one in Figure 9. These considerations support the (informal) argument that the new technique embodies an optimal trade-off between power and simplicity.

3.5. Implementation issues

While the technique is very easy to implement in a naïve way, it is legitimate to ask whether carrying around sets of pairs $\langle vt, st \rangle$ incurs severe penalties in terms of execution speed and memory occupation. Certainly, the new technique is less efficient than that in [3, Section 4.9.2]. However, in order to accept programs like that in Figure 3 some overhead due to the more elaborate analysis is unavoidable. For example, keeping track of the variables modified inside subroutines [3, Section 4.9.6] requires bit vectors or similar structures to be carried around.

The number of different calling addresses in a program is limited by the size of the program. No arithmetic is possible on calling addresses, which are generated by jsr and can only be moved around by the other instructions. Therefore, the sets cannot get exceedingly large (e.g. like integers). Experimental measures [9,19] suggest that current compilers generate code with very infrequent use of subroutines. So, in practice, the sets should be fairly small.

Anyhow, the following optimization is possible. The need to carry around sets of pairs arises in order to separate them at the ret instructions. If two singleton sets are merged that do not both contain calling addresses in the same variable or stack element, then the two pairs will never be separated and so can be merged into a single pair (extending *Type* with any). In other words, a hybrid merging strategy can be used: if pairs cannot be separated later, they are just merged into one pair, while they are kept distinct if there are different calling addresses in corresponding positions. So, if a program has no subroutines, all sets are singletons and the new technique essentially reduces to that in [3, Section 4.9.2]. The pairs of a set can lose their calling addresses via a store or pop instruction, in which case the transfer functions

**The requirement for the stack to have a fixed size for each address (given in [3] but unnecessary for type safety) may support more efficient execution of bytecode by means of native machine instructions as explained in [18]. In that case, it is trivial to extend the technique to check that all pairs $\langle vt, st \rangle$ in a set assigned to an address have stacks of the same size.

```
static int m(boolean x) {
  int y;
  try {
    if (x) return 1;
    y = 2;
  } finally {
    if (x) y = 3;
  }
  return y;
}
```

Figure 11. Type-safe Java code rejected by most techniques and verifiers.

for store and pop could merge the pairs into one. There are trade-offs related to having these transfer functions check and eventually merge pairs, as opposed to avoiding such additional checks in these transfer functions and unnecessarily keeping some pairs distinct when they could have been merged. These trade-offs should be evaluated experimentally.

Experimental work is necessary to more precisely evaluate the efficiency of the new technique, including optimizations. Existing programs may not constitute very meaningful test data because of their infrequent use of subroutines. Ideally, test data should include (future) bytecode programs that make more extensive and sophisticated use of subroutines.

4. RELATED WORK

All of the techniques to verify subroutines proposed in the literature approximate all integers with one type. So, their soundness could be proved using the integer-insensitive operational semantics. Therefore, by Theorem 2, every program verified by any of them is also verified by the new technique[††].

As a point of comparison among the techniques, consider the Java code in Figure 11 (adapted from [17, Figure 16.8]). The variable `y`, which contains an undefined value at the beginning of the method `m`, is definitely assigned a value before it is used by the `return`. Definite assignment is part of type safety and must be checked by bytecode verification in the JVM. The bytecode produced by mundane compilers (e.g. [6]) from Figure 11 is accepted by the new technique, as shown in Figure 12, where the real Java bytecode instructions (not those of $\mathcal{L}$) are used [3, Ch. 6], the exception handler for the `try` block [3, Section 7.12] is omitted, the variables are denoted by names instead of numbers and the type udf indicates that a variable contains an undefined value.

As another point of comparison, consider the Java code in Figure 13 (adapted from [9, Figure 6]). The `continue` inside the `finally` block, if executed, transfers control to the beginning of the

[††]Actually, some techniques [9,10,20] accept certain programs where the stack can grow without limit. If the limit *max* to the stack size is removed from $\mathcal{L}$, then the iterative data flow analysis of the new technique may not converge because the lattice becomes infinite. However, the per-method static limit on the stack is part of the semantics of Java bytecode, designed to relieve the execution engine from checking stack overflows.

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$	$vt_i(z)$	$vt_i(w)$
0	iload x	[ ]	int	udf	udf	udf
1	ifeq 7	[int]	int	udf	udf	udf
2	iconst_1	[ ]	int	udf	udf	udf
3	istore z	[int]	int	udf	udf	udf
4	jsr 11	[ ]	int	udf	int	udf
5	iload z	[ ]	int | int	udf | int	int | int	ca_4 | ca_4
6	ireturn	[int | int]	int | int	udf | int	int | int	ca_4 | ca_4
7	iconst_2	[ ]	int	udf	udf	udf
8	istore y	[int]	int	udf	udf	udf
9	jsr 11	[ ]	int	int	udf	udf
10	goto 17	[ ]	int	int	udf	ca_9
11	astore w	[ca_4 | ca_9]	int | int	udf | int	int | udf	udf | udf
12	iload x	[ ]	int | int	udf | int	int | udf	ca_4 | ca_9
13	ifeq 16	[int | int]	int | int	udf | int	int | udf	ca_4 | ca_9
14	iconst_3	[ ]	int | int	udf | int	int | udf	ca_4 | ca_9
15	istore y	[int | int]	int | int	udf | int	int | udf	ca_4 | ca_9
16	ret w	[ ]	int | int | int	udf | int | int	int | int | udf	ca_4 | ca_4 | ca_9
17	iload y	[ ]	int	int	udf	ca_9
18	ireturn	[int]	int	int	udf	ca_9

Figure 12. Successful verification of the bytecode for Figure 11.

```
static void m(boolean x) {
  while (x) {
    try {
      x = false;
    } finally {
      if (x) continue;
    }
  }
}
```

Figure 13. Type-safe Java code rejected by some techniques and verifiers.

`while` loop. The bytecode produced by mundane compilers (e.g. [6]) from Figure 13 is accepted by the new technique, as shown in Figure 14. Note that the subroutine, whose address range is 5–9, can be exited implicitly (i.e. not via a ret) from address 8, thus realizing the semantics of `continue`.

The official technique to verify subroutines [3,6] rejects the code in Figure 12. The types int and udf for y are merged into udf inside the subroutine (in that technique, udf coincides with any). Since y

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$
0	goto 10	[ ]	int	udf
1	iconst_0	[ ]	int | int	udf | ca_3
2	istore x	[int | int]	int | int	udf | ca_3
3	jsr 5	[ ]	int | int	udf | ca_3
4	goto 10	[ ]	int	ca_3
5	astore y	[ca_3 | ca_3]	int | int	udf | ca_3
6	iload x	[ ]	int	ca_3
7	ifeq 9	[int]	int	ca_3
8	goto 10	[ ]	int	ca_3
9	ret y	[ ]	int	ca_3
10	iload x	[ ]	int | int	udf | ca_3
11	ifne 1	[int | int]	int | int	udf | ca_3
12	return	[ ]	int | int	udf | ca_3

Figure 14. Successful verification of the bytecode for Figure 13.

may be modified at address 15, udf is propagated from address 16 to both 5 and 10, and thus eventually to 17, where iload y causes verification to fail, because it requires an int in y.

Stata and Abadi [11] presented the first formal technique to verify subroutines. Their work has been very influential, laying the foundations for further formal work in bytecode verification and subroutines; for instance, their original instruction set and operational semantics are used (sometimes with small modifications) in many publications, including this paper. They formalize verification as typing rules that can be implemented via data flow analysis. Similarly to [3,6], they keep track of the variables modified inside subroutines and selectively propagate types from ret and jsr, thus rejecting the code in Figure 12. They also impose a strict last-in, first-out use of subroutines.

Freund and Mitchell [7] generalize [11] to a more liberal use of subroutines, e.g. a subroutine may return to an outer caller, skipping one or more inner callers. They also study the interaction of subroutines with exceptions and object initialization. Their technique rejects the code in Figure 12 as well.

Qian [16] also gives typing rules that can be implemented via data flow analysis. He also keeps track of modified variables and selectively propagates types, thus rejecting the code in Figure 12. He records called subroutines using graphs instead of stacks, thus allowing a fairly liberal use of subroutines.

Stärk *et al.* [17] also use data flow analysis, keeping track of modified variables and selectively propagating types, thus rejecting the code in Figure 12[‡‡]. Since they impose almost no structure on the use of jsr and ret, their technique is relatively simple.

Hagiya and Tozawa [8] also give typing rules that can be implemented via data flow analysis. Types from callers are turned into special types before being propagated to subroutines; the special

[‡‡]Although they prove a theorem stating that their bytecode verifier accepts all the code produced by compilation, their notion of compilation does not fully capture existing compilers: the rules of definite assignment in [17, Section 8.3] expressly differ from those in [2, Ch. 16].

types reference variables at the callers so that no information is lost when types are merged inside subroutines. Upon subroutine return, the types are restored from the variables at the callers. This approach accepts subroutines that polymorphically swap the contents of two variables, which are rejected by the techniques above. A relatively liberal use of subroutines is allowed, e.g. a subroutine may return to an outer caller. Nevertheless, the code in Figure 12 is rejected because udf is assigned to y inside the subroutine and propagated to the caller's successor.

Leroy [21] describes a polyvariant data flow analysis for subroutines, employed in the off-card verifier of [22], developed by Trusted Logic. Subroutines are analyzed in different contexts for different callers; contexts include subroutine call stacks, which are extended by jsr and shrunk by ret. Almost no structure is imposed on the use of jsr and ret. While the code in Figure 12 is accepted, the code in Figure 14 is rejected. The technique includes checks for non-recursive calls to subroutines, as required by [3, Section 4.8.2]: these checks fail for the code in Figure 14, because the path out of the subroutine and back to address 1 propagates the call stack with the subroutine to address 3, where a false recursive call is detected.

Posegga and Vogt [23] advocate the use of model checking to exhaustively explore all the abstract execution states of a program. Their abstract states include subroutine call stacks, which are extended by jsr and shrunk by ret. The abstract execution of jsr includes checks for non-recursive subroutine calls [3, Section 4.8.2], which fail for the code in Figure 14, similarly to [21] described above. If subroutine call stacks and their relative checks are removed, then the abstract state exploration is as powerful as the new technique, because the abstract values are exactly the types of the concrete values. However, this approach is meant for off-line verification, because deploying a model checker within the JVM is problematic.

As evidenced by the new technique, recursive subroutine calls are harmless to type safety. The prescription in [3, Section 4.8.2] prohibiting recursive subroutine calls is not only unnecessary, but also misleading, as manifested by the last two examples. Interestingly, the verifier in [6] accepts the code in Figure 14 because it merges subroutine call stacks by computing their common sub-stacks; so, at address 10 the non-empty stack from 8 is merged with the empty stack from 0 resulting in the empty stack, which is propagated back to 1 and eventually to 3, with no false recursion being detected.

Yelland [24] proposes an encoding of Java bytecode in the functional programming language Haskell [25] such that bytecode verification boils down to Haskell's type checking. Unfortunately, the paper does not provide full details about the treatment of subroutines, but it seems comparable in power with the new technique, e.g. it should accept the code in Figures 12 and 14. This encoding of Java bytecode in Haskell is useful as a specification that integrates operational semantics and bytecode verification, but it is not straightforward to use or develop a Haskell type checker as a bytecode verifier inside a JVM implementation. In addition, this approach is less accessible, compared to data flow analysis, to practitioners and researchers unfamiliar with Haskell and monads. Lastly, no simple characterization of accepted programs is provided.

O'Callahan [9] presents declarative rules for type assignments. Types include variables to capture polymorphism. A return address (i.e. a successor of a calling address) is assigned a continuation type that embeds variable and stack types for the return address. The rules impose almost no structure on the use of jsr and ret. His technique is comparable in power with the new technique, e.g. it accepts the code in Figures 12 and 14. However, while it is not hard to check whether a given type assignment satisfies the declarative rules, it is not immediate how to compute such an assignment. In addition, no simple characterization of accepted programs is provided.

Retert and Boyland [10] apply a general framework for interprocedural analysis to the verification of Java bytecode subroutines. The behavior of each subroutine is captured by sets of input–output pairs of type assignments to the variables and stack, computed via a fixed-point iteration where old pairs for callees are used to compute new pairs for callers. Subroutine 'boundaries' are implicitly computed by considering ret and certain occurrences of jsr as endpoints (the latter serve to cope with the implicit exiting of subroutines as in Figure 14). The power of their technique approaches the new technique, e.g. it accepts the code in Figures 12 and 14. It cannot merge stacks of different sizes unless the merging takes place at a subroutine address (e.g. a program without subroutines where the stack may have different sizes at certain addresses is rejected), but this is probably not a problem in practice, given the argument in [18] mentioned in Section 3.4. However, their approach is more complicated than the new technique. In addition, no simple characterization of accepted programs is provided.

As reported in [21], Brisset [26] independently discovered the new technique, formalized it using the Coq proof assistant [27], and extracted an implementation, in the functional programming language ML [28], from the correctness proof. However, he did not publish his work in any paper.

Henrio and Serpette [20] also independently discovered the new technique. They present a framework for bytecode verification where multiple abstract execution states (i.e. states where values are abstracted to types) can be assigned to addresses and can be reduced by applying a merging operation. Depending on the choice of the merging operation, the resulting algorithm ranges from the new technique (if merging is identity) to [3, Section 4.9.2] (if multiple states are always merged into one). However, their description of the new technique is not as thorough as the one in this paper; in particular, explicit proofs and the characterization of accepted programs are not given.

There exist several commercial and academic implementations of the JVM which include bytecode verifiers, but no documentation is readily available about their treatment of subroutines. Anyhow [17, Section 16.1.1] reports that the code in Figure 12 is rejected by all the verifiers tried by the authors, including those in various versions of Netscape and Internet Explorer, as well as the Kimera verifier [29]. These verifiers all probably keep track of modified variables and selectively propagate types like [3,6].

As part of the OVM project [30], Grothoff [31] independently implemented a verifier whose treatment of subroutines is the same as the new technique.

Trusted Logic's on-terminal verifier [32] (for the JEFF file format [33]), independently implemented by Frey [34], also treats subroutines in the same way; this verifier is very space- and time-efficient, thus demonstrating the practicality of the technique.

Klein and Wildmoser [35] formalize and mechanically prove the soundness of the new technique using the Isabelle/HOL theorem prover [36].

Wildmoser [37] formalizes the techniques in [7,11,21] and this paper in a common formalism and formally compares their relative power (i.e. sets of accepted programs). He also proposes an algorithm to in-line subroutines, proving that it preserves program equivalence and calculates a bound on the size of the expanded code.

The optimization of the new technique informally described in Section 3.5 has been independently formalized in [20] (as a possible choice of the merging operation) and has been proved to be equivalent to the new technique (in the sense that it accepts the same programs) in [21]. It is also reported in [21] that Frey independently implemented that optimization.

It has been argued [19] that the little space saved by subroutines in typical code does not warrant the increased complexity in verification. However, subroutines cannot easily be eliminated, for backward

compatibility reasons. In addition, the space saved by subroutines could be relevant in memory-constrained devices such as smart cards [38].

A way around the problem of verifying subroutines could be to rewrite bytecode prior to verification, in order to in-line subroutines as prescribed in [39] or in order to split variables to make their types unique as described in [40]. After the rewriting, the technique in [3, Section 4.9.2] can be used. However, since subroutines can be exited implicitly as in Figure 14, their boundaries may not be always easy to determine. So, it is unclear whether bytecode rewriting is a simpler problem than verification with subroutines; for instance, Wildmoser's algorithm mentioned above is relatively complex, compared with the new technique. It is also unclear whether rewriting followed by the simpler analysis is more efficient than just the slightly more complex analysis of the new technique without rewriting; experimental measures are needed. In addition, variable splitting may require the introduction of additional instructions at the beginning of a method to ensure that all variables are defined [19], causing some loss of performance.

A short version of this paper is available as [41], while [15] is a more comprehensive paper on the topic of subroutines. In [12] the new technique is lifted to a complete formalization of Java bytecode verification. In 2000, the author used Specware [42], a system for the formal specification and refinement of software, to derive a bytecode verifier from the formalization in [12]; this verifier can serve as a high-assurance reference implementation*.

The language $\mathcal{L}$ not only abstracts Java bytecode but also other assembly-like languages, e.g. machine languages of microprocessors. Many kinds of static analysis can be cast into a data flow analysis framework where information about values is captured as 'types', e.g. security levels for information flow analysis. So, the new technique may have a broader applicability than Java bytecode verification.

Despite the similarities between the JVM and the .NET Common Language Runtime (CLR) [43], the latter does not feature subroutines. The main purpose of subroutines in the JVM is to realize the semantics of Java's `finally`; in contrast, the CLR has a built-in `finally` construct.

APPENDIX A

Proof of Lemma 1

Since $\mathit{Verified}(P)$ holds, $\sigma_0 \neq \mathsf{fail}$. Since $\sigma_0 \sqsupseteq l_{\mathsf{Init}}$, $\langle \lambda x.\mathsf{int}, [\,] \rangle \in \sigma_0$. Since $\alpha(\lambda x.0_\mathrm{I}) = \lambda x.\mathsf{int}$ and $\alpha([\,]) = [\,]$, $\mathit{Inv}(\mathsf{Init})$ holds.

Proof of Lemma 2

Let $stt = \langle i, vr, sk \rangle$. Since $\mathit{Inv}(stt)$ holds, $\langle \alpha(vr), \alpha(sk) \rangle \in \sigma_i$. The proof that $\mathit{Inv}(stt')$ holds is by case analysis on P_i.

If $P_i = \mathsf{push0}$, then $i + 1 \in \mathcal{D}(P)$ by the third requirement on programs in Section 2.1. Since $\mathit{Verified}(P)$ holds, $\sigma_{i+1} \neq \mathsf{fail}$. Since $\sigma_{i+1} \sqsupseteq tf_{\mathsf{push0}}(\sigma_i)$, $|\alpha(sk)| < max$ and $\langle \alpha(vr), \alpha(sk) \cdot \mathsf{int} \rangle \in \sigma_{i+1}$. Since $|sk| < max$, by rule (PH) $stt' = \langle i + 1, vr, sk \cdot 0_\mathrm{I} \rangle$. Since $\alpha(sk \cdot 0_\mathrm{I}) = \alpha(sk) \cdot \mathsf{int}$, $\mathit{Inv}(stt')$ holds.

*However, it is not presently publicly available.

The cases $P_i = \mathsf{inc}$, $P_i = \mathsf{div}$, $P_i = \mathsf{pop}$, $P_i = \mathsf{load}\, x$, $P_i = \mathsf{store}\, x$, $P_i = \mathsf{if0}\ j$ and $P_i = \mathsf{jsr}\ s$ are proved analogously.

If $P_i = \mathsf{ret}\, x$, then $C \neq \emptyset$ by the fourth requirement on programs in Section 2.1. By the third requirement on programs in Section 2.1, $c + 1 \in \mathcal{D}(P)$ for all $c \in C$. Since $\mathit{Verified}(P)$ holds, $\sigma_{c+1} \neq \mathsf{fail}$ for all $c \in C$. Since $\sigma_{c+1} \sqsupseteq \mathit{tf}^{x,c}_{\mathsf{ret}}(\sigma_i)$ for all $c \in C$ (there exists at least one because $C \neq \emptyset$), $\alpha(vr)(x) = \mathsf{ca}_{c'}$ for a specific $c' \in C$ and $\langle \alpha(vr), \alpha(sk)\rangle \in \sigma_{c'+1}$ (by the definition of $\mathit{tf}^{x,c'}_{\mathsf{ret}}$). Since $vr(x) = c'_{\mathrm{C}}$, by rule (RT) $stt' = \langle c' + 1, vr, sk\rangle$. Since $\langle \alpha(vr), \alpha(sk)\rangle \in \sigma_{c'+1}$, $\mathit{Inv}(stt')$ holds.

The case $P_i = \mathsf{halt}$ is impossible because it contradicts $stt \rightsquigarrow stt'$.

Proof of Theorem 1 (Soundness)

The proof is by contradiction. Suppose that $\mathit{TypeSafe}(P)$ does not hold. Then $\mathsf{Init} \rightsquigarrow^{+} \mathsf{Err}$, i.e. there exist $stt_0, \ldots, stt_n$ such that $n \geq 1$, $stt_0 = \mathsf{Init}$, $stt_n = \mathsf{Err}$ and $stt_{k-1} \rightsquigarrow stt_k$ for $1 \leq k \leq n$. By Lemma 1, $\mathit{Inv}(stt_0)$ holds. By Lemma 2 and induction on k, $\mathit{Inv}(stt_k)$ holds for $1 \leq k \leq n$. However, $\mathit{Inv}(stt_n)$ contradicts $stt_n = \mathsf{Err}$: therefore, $\mathit{TypeSafe}(P)$ must hold.

Proof of Theorem 2 (Characterization)

The implication $\mathit{Verified}(P) \Rightarrow \mathit{TypeSafe}_{\mathrm{I}}(P)$ is proved in complete analogy with Theorem 1 (including Lemmas 1 and 2). The implication $\mathit{TypeSafe}_{\mathrm{I}}(P) \Rightarrow \mathit{Verified}(P)$ is proved by constructing an assignment γ of lattice elements to addresses such that fail does not appear in γ and then showing that γ is a solution to the data flow analysis constraints. Since $\gamma \sqsupseteq \sigma$, fail does not appear in σ and therefore $\mathit{Verified}(P)$ holds.

The assignment γ is defined by

$$\gamma_i = \{\langle \alpha(vr), \alpha(sk)\rangle \mid \mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i, vr, sk\rangle\}.$$

In other words, the value of γ_i is given by abstracting all possible states of execution with program counter i that are reachable from Init. Each γ_i is finite, because Type, VN and max are finite. Each $\gamma_i \neq \mathsf{fail}$ by construction.

Consider the constraint $u_0 \sqsupseteq l_{\mathsf{Init}}$: since $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \mathsf{Init}$, $\langle \lambda x.\mathsf{int}, [\,]\rangle \in \gamma_0$ and therefore $\gamma_0 \sqsupseteq l_{\mathsf{Init}}$. Consider a constraint $u_{i'} \sqsupseteq \mathit{tf}(u_i)$: the proof that $\gamma_{i'} \sqsupseteq \mathit{tf}(\gamma_i)$ is by case analysis on P_i.

If $P_i = \mathsf{push0}$, then $\mathit{tf} = \mathit{tf}_{\mathsf{push0}}$ and $i' = i + 1$. Consider an arbitrary $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i, vr, sk\rangle$. Since $\mathit{TypeSafe}_{\mathrm{I}}(P)$ holds, $|sk| < \mathit{max}$ and so $|\alpha(sk)| < \mathit{max}$. Since $\langle \alpha(vr), \alpha(sk)\rangle$ is arbitrary, $\mathit{tf}(\gamma_i) \neq \mathsf{fail}$. Consider an arbitrary $\langle vt, st\rangle \in \mathit{tf}(\gamma_i)$. By the definition of $\mathit{tf}_{\mathsf{push0}}$, there must exist $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i, vr, sk\rangle$, such that $vt = \alpha(vr)$ and $st = \alpha(sk) \cdot \mathsf{int}$. By rule (PH), $\langle i, vr, sk\rangle \rightsquigarrow_{\mathrm{I}} \langle i', vr, sk \cdot 0_{\mathrm{I}}\rangle$ and so $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i', vr, sk \cdot 0_{\mathrm{I}}\rangle$: therefore, $\langle \alpha(vr), \alpha(sk \cdot 0_{\mathrm{I}})\rangle \in \gamma_{i'}$. Since $\alpha(sk \cdot 0_{\mathrm{I}}) = st$, $\langle vt, st\rangle \in \gamma_{i'}$. Since $\langle vt, st\rangle$ is arbitrary, $\gamma_{i'} \sqsupseteq \mathit{tf}(\gamma_i)$.

The cases $P_i = \mathsf{inc}$, $P_i = \mathsf{div}$, $P_i = \mathsf{pop}$, $P_i = \mathsf{load}\, x$, $P_i = \mathsf{store}\, x$ and $P_i = \mathsf{jsr}\ s$ are proved analogously.

If $P_i = \mathsf{if0}\ j$, then $\mathit{tf} = \mathit{tf}_{\mathsf{if0}}$ and $i' \in \{i + 1, j\}$. Consider an arbitrary $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i, vr, sk\rangle$. Since $\mathit{TypeSafe}_{\mathrm{I}}(P)$ holds, $sk = sk' \cdot \iota_{\mathrm{I}}$ for some sk' and ι, and so $\alpha(sk) = \alpha(sk') \cdot \mathsf{int}$. Since $\langle \alpha(vr), \alpha(sk)\rangle$ is arbitrary, $\mathit{tf}(\gamma_i) \neq \mathsf{fail}$. Consider an arbitrary $\langle vt, st\rangle \in \mathit{tf}(\gamma_i)$. By the definition of $\mathit{tf}_{\mathsf{if0}}$, there must exist $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}^{*} \langle i, vr, sk\rangle$, such that $vt = \alpha(vr)$ and $\alpha(sk) = st \cdot \mathsf{int}$, which implies $sk = sk' \cdot \iota_{\mathrm{I}}$ for some sk' and ι such that $\alpha(sk') = st$.

If ($\iota = 0 \wedge i' = j$) or ($\iota \neq 0 \wedge i' = i + 1$), then by rule (IF) $\langle i, vr, sk\rangle \rightsquigarrow_{\mathrm{I}} \langle i', vr, sk'\rangle$. If ($\iota = 0 \wedge i' = i + 1$) or ($\iota \neq 0 \wedge i' = j$), then by rule (IF') $\langle i, vr, sk\rangle \rightsquigarrow_{\mathrm{I}} \langle i', vr, sk'\rangle$. (Note that without rule (IF'), this proof would not work. In fact, as shown by the example in Figure 9, $\mathit{TypeSafe}(P) \not\Rightarrow \mathit{Verified}(P)$.) In both cases, $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}{}^{*} \langle i', vr, sk'\rangle$: therefore, $\langle \alpha(vr), \alpha(sk')\rangle \in \gamma_{i'}$. Since $\alpha(sk') = st$, $\langle vt, st\rangle \in \gamma_{i'}$. Since $\langle vt, st\rangle$ is arbitrary, $\gamma_{i'} \sqsupseteq tf(\gamma_i)$.

If $P_i = \mathsf{ret}\ x$, then $tf = tf^{x,c}_{\mathsf{ret}}$ and $i' = c + 1$ for some $c \in C$. Consider an arbitrary $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}{}^{*} \langle i, vr, sk\rangle$. Since $\mathit{TypeSafe}_{\mathrm{I}}(P)$ holds, $vr(x) = c'_{\mathrm{C}}$ for some c' and so $\alpha(vr)(x) = \mathsf{ca}_{c'}$. Since $\langle \alpha(vr), \alpha(sk)\rangle$ is arbitrary, $tf(\gamma_i) \neq \mathsf{fail}$. Consider an arbitrary $\langle vt, st\rangle \in tf(\gamma_i)$. By the definition of $tf^{x,c}_{\mathsf{ret}}$, there must exist $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}{}^{*} \langle i, vr, sk\rangle$, such that $vt = \alpha(vr)$, $st = \alpha(sk)$ and $vt(x) = \mathsf{ca}_c$ (which implies $vr(x) = c_{\mathrm{C}}$). By rule (RT), $\langle i, vr, sk\rangle \rightsquigarrow_{\mathrm{I}} \langle i', vr, sk\rangle$ and so $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}{}^{*} \langle i', vr, sk\rangle$: therefore, $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_{i'}$, which means $\langle vt, st\rangle \in \gamma_{i'}$. Since $\langle vt, st\rangle$ is arbitrary, $\gamma_{i'} \sqsupseteq tf(\gamma_i)$.

The case $P_i = \mathsf{halt}$ is impossible because $\mathit{IConstr}(i, \mathsf{halt}) = \emptyset$.

Aside: Proof that the constructed solution is the least one

This is the case that $\gamma = \sigma$, which is proved as follows. Recall that fail does not appear in σ. Consider an arbitrary $i \in \mathcal{D}(P)$ and an arbitrary $\langle \alpha(vr), \alpha(sk)\rangle \in \gamma_i$, where $\mathsf{Init} \rightsquigarrow_{\mathrm{I}}{}^{*} \langle i, vr, sk\rangle$. So, there exist $stt_0, \ldots, stt_n$ such that $n \geq 0$, $stt_0 = \mathsf{Init}$, $stt_n = \langle i, vr, sk\rangle$, and $stt_{k-1} \rightsquigarrow_{\mathrm{I}} stt_k$ for $1 \leq k \leq n$. Let $stt_k = \langle i_k, vr_k, sk_k\rangle$ for $0 \leq k \leq n$. So, there is a data flow analysis constraint $u_{i_{k+1}} \sqsupseteq tf_k(u_{i_k})$ for $0 \leq k < n$ (easily proved by case analysis on P_{i_k}). Since σ is a solution, $\sigma_{i_0} \sqsupseteq l_{\mathsf{Init}}$ and $\sigma_{i_{k+1}} \sqsupseteq tf_k(\sigma_{i_k})$ for $0 \leq k < n$. By induction on k (below), $\langle \alpha(vr_k), \alpha(sk_k)\rangle \in \sigma_{i_k}$ for $0 \leq k \leq n$. So, in particular, $\langle \alpha(vr), \alpha(sk)\rangle \in \sigma_i$. Since $\langle \alpha(vr), \alpha(sk)\rangle$ is arbitrary, $\gamma_i \sqsubseteq \sigma_i$ and therefore $\gamma_i = \sigma_i$. Since i is arbitrary, $\gamma = \sigma$.

The base case of the induction is proved by noting that $\langle \alpha(vr_0), \alpha(sk_0)\rangle \in l_{\mathsf{Init}}$ and therefore $\langle \alpha(vr_0), \alpha(sk_0)\rangle \in \sigma_{i_0}$ because $\sigma_{i_0} \sqsupseteq l_{\mathsf{Init}}$. The step case of the induction assumes that $\langle \alpha(vr_k), \alpha(sk_k)\rangle \in \sigma_{i_k}$, with $0 \leq k < n$: the fact that $\langle \alpha(vr_{k+1}), \alpha(sk_{k+1})\rangle \in \sigma_{i_{k+1}}$ is proved by case analysis on P_{i_k}.

If $P_{i_k} = \mathsf{push0}$, then $tf_k = tf_{\mathsf{push0}}$. Since $stt_k \rightsquigarrow_{\mathrm{I}} stt_{k+1}$, by rule (PH) $vr_{k+1} = vr_k$ and $sk_{k+1} = sk_k \cdot 0_{\mathrm{I}}$. By the definition of tf_{push0}, $\langle \alpha(vr_k), \alpha(sk_k) \cdot \mathsf{int}\rangle \in tf_k(\sigma_{i_k})$. Since $\alpha(sk_{k+1}) = \alpha(sk_k) \cdot \mathsf{int}$, $\langle \alpha(vr_{k+1}), \alpha(sk_{k+1})\rangle \in \sigma_{i_{k+1}}$ because $\sigma_{i_{k+1}} \sqsupseteq tf_k(\sigma_{i_k})$.

The cases $P_{i_k} = \mathsf{inc}$, $P_{i_k} = \mathsf{div}$, $P_{i_k} = \mathsf{pop}$, $P_{i_k} = \mathsf{load}\ x$, $P_{i_k} = \mathsf{store}\ x$, $P_{i_k} = \mathsf{if0}\ j$ and $P_{i_k} = \mathsf{jsr}\ s$ are proved analogously.

If $P_{i_k} = \mathsf{ret}\ x$, then $tf_k = tf^{x,c}_{\mathsf{ret}}$, where $c + 1 = i_{k+1}$. Since $stt_k \rightsquigarrow_{\mathrm{I}} stt_{k+1}$, by rule (RT) $vr_{k+1} = vr_k$, $sk_{k+1} = sk_k$ and $vr_k(x) = c$. By the definition of $tf^{x,c}_{\mathsf{ret}}$, $\langle \alpha(vr_k), \alpha(sk_k)\rangle \in tf_k(\sigma_{i_k})$ because $\alpha(vr_k)(x) = \mathsf{ca}_c$. So, $\langle \alpha(vr_{k+1}), \alpha(sk_{k+1})\rangle \in \sigma_{i_{k+1}}$ because $\sigma_{i_{k+1}} \sqsupseteq tf_k(\sigma_{i_k})$.

The case $P_{i_k} = \mathsf{halt}$ is impossible because it contradicts $stt_k \rightsquigarrow_{\mathrm{I}} stt_{k+1}$.

REFERENCES

1. Arnold K, Gosling J, Holmes D. *The Java™ Programming Language* (3rd edn). Addison-Wesley: Reading, MA, 2000.
2. Gosling J, Joy B, Steele G, Bracha G. *The Java™ Language Specification* (2nd edn). Addison-Wesley: Reading, MA, 2000.
3. Lindholm T, Yellin F. *The Java™ Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, 1999.

4. Yellin F. Low level security in Java. *Proceedings of the 4th International World Wide Web Conference*. O'Reilly & Associates, 1995; 369–379. http://java.sun.com/sfaq/verifier.html.
5. Gong L. *Inside Java*TM *2 Platform Security*. Addison-Wesley: Reading, MA, 1999.
6. Sun Microsystems. Java 2 SDK Standard Edition version 1.4. http://java.sun.com/j2se.
7. Freund S, Mitchell J. A type system for Java bytecode subroutines and exceptions. *Technical Note STAN-CS-TN-99-91*, Computer Science Department, Stanford University, August 1999.
8. Hagiya M, Tozawa A. On a new method for dataflow analysis of Java virtual machine subroutines. *Proceedings of the 5th Static Analysis Symposium (SAS'98)* (*Lecture Notes in Computer Science*, vol. 1503). Springer: Berlin, 1998; 17–32.
9. O'Callahan R. A simple, comprehensive type system for Java bytecode subroutines. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, January 1999. ACM Press, 1999; 70–78.
10. Retert W, Boyland J. Interprocedural analysis for JVML verification. *Proceedings of the 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002.
11. Stata R, Abadi M. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems* 1999; **21**(1):90–137.
12. Coglio A. Java bytecode verification: A complete formalization. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
13. Coglio A. Improving the official specification of Java bytecode verification. *Proceedings of the 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.
14. Nielson F, Nielson HR, Hankin C. *Principles of Program Analysis*. Springer: Berlin, 1998.
15. Coglio A. Java bytecode subroutines demystified. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
16. Qian Z. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. *Formal Syntax and Semantics of Java* (*Lecture Notes in Computer Science*, vol. 1523), Alves-Foss J (ed.). Springer: Berlin, 1999; 271–312.
17. Stärk R, Schmid J, Börger E. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer: Berlin, 2001.
18. Gosling J. Java intermediate bytecode. Proceedings of the Workshop on Intermediate Representations (IR'95). *ACM SIGPLAN Notices* 1995; **30**(3):111–118.
19. Freund S. The costs and benefits of Java bytecode subroutines. *Proceedings of the OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
20. Henrio L, Serpette B. A parametrized polyvariant bytecode verifier. *Proceedings of the Journées Francophones des Langages Applicatifs (JFLA'03)*, January 2003.
21. Leroy X. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*. To appear.
22. Sun Microsystems. Java Card Development Kit version 2.1.2. http://java.sun.com/javacard.
23. Posegga J, Vogt H. Java bytecode verification using model checking. *Proceedings of the OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
24. Yelland P. A compositional account of the Java virtual machine. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*. ACM Press, 1999; 57–69.
25. The Haskell Web site. http://www.haskell.org.
26. Brisset P. Vers un vérifieur de bytecode Java certifié. *Seminar Given at Ecole Normale Supérieure Paris*, October 1998.
27. The Coq proof assistant. http://coq.inria.fr.
28. Milner R, Tofte M, Harper R, MacQueen D. *The Definition of Standard ML*. MIT Press: Cambridge, MA, 1997.
29. The Kimera project Web site. http://kimera.cs.washington.edu.
30. The OVM project Web site. http://ovmj.org.
31. Grothoff C. Private communication, June 2001.
32. Trusted Logic. TL Embedded Verifier. http://www.trusted-logic.fr/solution/TL_Embedded_Verifier.html.
33. J Consortium. JEFF™ file format. http://www.j-consortium.org [2002].
34. Frey A. Private communication, May 2002.
35. Klein G, Wildmoser M. Verified bytecode subroutines. *Journal of Automated Reasoning*. To appear.
36. The Isabelle system. http://isabelle.in.tum.de.
37. Wildmoser M. Subroutines and Java bytecode verification. *Master's Thesis*, Technical University of Munich, June 2002.
38. Chen Z. *Java Card*TM *Technology for Smart Cards*. Addison-Wesley: Reading, MA, 2000.
39. Sun Microsystems. Connected, limited device configuration: Specification version 1.0a. http://java.sun.com/j2me [May 2000].
40. Agesen O, Detlefs D, Eliot J, Moss B. Garbage collection and local variable type-precision and liveness in Java virtual machines. Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation (PLDI'98). *ACM SIGPLAN Notices* 1998; **33**(5):269–279.
41. Coglio A. Simple verification technique for complex Java bytecode subroutines. *Proceedings of the 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002.
42. Kestrel Institute and Kestrel Technology LLC. Specware™. http://www.specware.org.
43. ECMA International. Common Language Infrastructure (CLI) Standard, ECMA-335, December 2002.

