Vocabulary of the defensive $\text{JVM}_{\mathcal{I}}$

Universes:

$$\begin{aligned} \mathit{WordType} = {} & \texttt{int} \\ & \mid \texttt{lowLong} \\ & \mid \texttt{highLong} \\ & \mid \texttt{float} \\ & \mid \texttt{lowDouble} \\ & \mid \texttt{highDouble} \end{aligned}$$

$$\mathit{Word} = (\mathit{32\text{-}Bitstring}, \mathit{WordType})$$

Extracting run-time type information:

$$type(\{(x_1, (w_1, t_1)), \ldots, (x_n, (w_n, t_n))\}) := \{(x_1, t_1), \ldots, (x_n, t_n)\}$$
$$type([(w_1, t_1), \ldots, (w_n, t_n)]) = [t_1, \ldots, t_n]$$

Type frames (stack maps):

$$(regT, opdT) = (type(reg), type(opd))$$

Defensive $\text{JVM}_{\mathcal{I}}$ = trustful $\text{JVM}_{\mathcal{I}}$ + run-time checks

$$
\begin{array}{l}
defensiveScheme_I(check, trustfulVM) = \\
\quad \textbf{if } \neg validCodeIndex(code, pc) \vee \\
\qquad \neg check(instr, maxOpd, pc, type(reg), type(opd)) \textbf{ then} \\
\qquad\quad halt := \texttt{"Runtime check failed"} \\
\quad \textbf{else} \\
\qquad trustfulVM
\end{array}
$$

$$
\begin{array}{l}
code = code(meth) \\
instr = code(meth)(pc) \\
maxOpd = maxOpd(meth)
\end{array}
$$

$$validCodeIndex(code, pc) = (0 \leq pc \wedge pc < length(code))$$

$$defensiveVM_I = defensiveScheme_I(check_I, trustfulVM_I)$$

Checking JVM$_{\mathcal{I}}$ instructions

$$
\begin{array}{l}
check_I(instr, maxOpd, pc, regT, opdT) = \\
\quad \textbf{case } instr \textbf{ of} \\
\qquad Prim(p) \rightarrow opdT \sqsubseteq_{\text{suf}} argTypes(p) \wedge \\
\qquad\qquad\qquad \neg over(maxOpd, opdT, retSize(p) - argSize(p)) \\
\qquad Dupx(s_1, s_2) \rightarrow \textbf{let } [ts_1, ts_2] = tops(opdT, [s_1, s_2]) \\
\qquad\qquad\qquad\qquad length(opdT) \geq s_1 + s_2 \wedge \\
\qquad\qquad\qquad\qquad \neg overflow(maxOpd, opdT, s_2) \wedge \\
\qquad\qquad\qquad\qquad validTypeSeq(ts_1) \wedge validTypeSeq(ts_2) \\
\qquad Pop(s) \rightarrow length(opdT) \geq s \\
\qquad Load(t, x) \rightarrow \\
\qquad\quad \textbf{if } size(t) = 1 \textbf{ then } [regT(x)] \sqsubseteq_{\text{mv}} t \wedge \neg over(maxOpd, opdT, 1) \\
\qquad\quad \textbf{else } [regT(x), regT(x+1)] \sqsubseteq_{\text{mv}} t \wedge \neg over(maxOpd, opdT, 2) \\
\qquad Store(t, \_) \rightarrow opdT \sqsubseteq_{\text{suf}} t \\
\qquad Goto(o) \rightarrow True \\
\qquad Cond(p, o) \rightarrow opdT \sqsubseteq_{\text{suf}} argTypes(p)
\end{array}
$$

Checking $\text{JVM}_{\mathcal{I}}$ instructions (continued)

Overflow of operand stack:

$$over(maxOpd, opdT, s) = length(opdT) + s > maxOpd$$

No splitting of double words:

$$\begin{aligned} validTypeSeq([\,]) &= True \\ validTypeSeq([t]) &= \neg isHigh(t) \\ validTypeSeq([t, \_]) &= \neg isHigh(t) \end{aligned}$$

$$isHigh(t) = (t = \texttt{highLong} \vee t = \texttt{highDouble})$$

Definition of $\sqsubseteq_{\text{mv}}$:

$$\begin{aligned} [\texttt{int}] &\sqsubseteq_{\text{mv}} \texttt{int} &&= True \\ [\texttt{float}] &\sqsubseteq_{\text{mv}} \texttt{float} &&= True \\ [\texttt{lowLong}, \texttt{highLong}] &\sqsubseteq_{\text{mv}} \texttt{long} &&= True \\ [\texttt{lowDouble}, \texttt{highDouble}] &\sqsubseteq_{\text{mv}} \texttt{double} &&= True \end{aligned}$$

Defensive JVM_C = trustful JVM_C + run-time checks

$$defensiveVM_C = defensiveScheme_C(check_C, trustfulVM_C)$$

$$defensiveScheme_C(check, trustfulVM) =$$
$$\quad \textbf{if } switch = Noswitch \textbf{ then}$$
$$\quad\quad defensiveScheme_I(check(meth), trustfulVM)$$
$$\quad \textbf{else}$$
$$\quad\quad trustfulVM$$

Checking JVM$_C$ instructions

$$
\begin{array}{l}
check_C(meth)(instr, maxOpd, pc, regT, opdT) = \\
\quad check_I(instr, maxOpd, pc, regT, opdT) \vee \\
\quad \textbf{case } instr \textbf{ of} \\
\qquad GetStatic(t, c/f) \quad \longrightarrow \neg overflow(maxOpd, opdT, size(t)) \\
\qquad PutStatic(t, c/f) \quad \longrightarrow opdT \sqsubseteq_{\text{suf}} t \\
\qquad InvokeStatic(t, c/m) \longrightarrow opdT \sqsubseteq_{\text{suf}} argTypes(c/m) \wedge \\
\qquad\qquad\qquad\qquad \neg overflow(maxOpd, opdT, \\
\qquad\qquad\qquad\qquad\qquad\qquad size(t) - argSize(c/m)) \\
\qquad Return(t) \quad\quad\quad\quad \longrightarrow opdT \sqsubseteq_{\text{suf}} returnType(meth) \wedge \\
\qquad\qquad\qquad\qquad returnType(meth) \sqsubseteq_{\text{mv}} t
\end{array}
$$

Refinement of $\sqsubseteq_{\text{mv}}$:

$$[\,] \sqsubseteq_{\text{mv}} \texttt{void} = True$$

Object initialization in the defensive JVM$_{\mathcal{O}}$

1. A newly created object is regarded as un-initialized. An object becomes fully initialized when the constructor of class `Object` is invoked. The invocation of another constructor makes an object partially initialized.
2. Constructors are invoked on un-initialized or partially initialized objects only.
3. Field accesses are performed on fully initialized objects only.
4. Instance methods are invoked on fully initialized objects only.
5. References to not fully initialized objects are neither stored in class fields, nor in instance fields, nor in array elements.
6. References to not fully initialized objects can be moved from the operand stack to local registers and vice versa. They can also be compared with other references using the operator '=='. The *Checkcast* and *Instanceof* instructions are applied to fully initialized objects only.

Verify types

VerifyType
$=$ `int`
| `lowLong`
| `highLong`
| `float`
| `lowDouble`
| `highDouble`
| `Null`
| *Class*
| *Interface*
| *Array*
| $(\mathit{Class}, \mathit{Pc})_{new}$
| InInit
| `unusable`

Definition. For verify types σ and τ the relation $\sigma \sqsubseteq \tau$ is true, iff one of the following conditions is true:

- $\sigma = \tau$, or
- σ and τ are reference types and $\sigma \preceq \tau$, or
- $\tau =$ `unusable`.

Vocabulary of the defensive JVM$_{\mathcal{O}}$

Refinement of $\sqsubseteq_{\mathrm{mv}}$:

$$\begin{array}{ll} [c] & \sqsubseteq_{\mathrm{mv}} \texttt{addr} = \mathit{True} \\ [(\_,\_)_{new}] & \sqsubseteq_{\mathrm{mv}} \texttt{addr} = \mathit{True} \\ [\mathit{InInit}] & \sqsubseteq_{\mathrm{mv}} \texttt{addr} = \mathit{True} \end{array}$$

$$\mathit{isHigh}(t) = (t = \texttt{highLong} \vee t = \texttt{highDouble} \vee t = \texttt{unusable})$$

Universe:

$$\mathit{InitState} = \mathit{New}(\mathit{Pc}) \mid \mathit{InInit} \mid \mathit{Complete}$$

New dynamic function:

$$\mathit{initState}\colon \mathit{Ref} \to \mathit{InitState}$$

Refinement of $\mathit{execVM}_{\mathcal{O}}$:

If $\mathit{execVM}_{\mathcal{O}}$ executes $\mathit{New}(c)$ and creates a new reference r:

$$\mathit{initState}(r) := \mathit{New}(\mathit{pc})$$

Refinement of $pushFrame$

$$
\begin{array}{l}
pushFrame(c/m, args) = \\
\quad stack := stack \cdot [(pc, reg, opd, meth)] \\
\quad meth := c/m \\
\quad pc \quad := 0 \\
\quad opd \quad := [\,] \\
\quad reg \quad := makeRegs(args) \\
\quad \textbf{if } methNm(m) = \texttt{"<init>"} \textbf{ then} \\
\qquad \textbf{let } [r] \cdot \_ = args \\
\qquad \textbf{if } c = \texttt{Object} \textbf{ then} \\
\qquad\quad initState(r) := Complete \\
\qquad \textbf{else} \\
\qquad\quad initState(r) := InInit
\end{array}
$$

Vocabulary of the defensive JVM$_{\mathcal{O}}$ (continued)

Refinement of WordType:

$$\mathit{WordType} = \ldots \mid \texttt{reference}$$

The type of a reference (pointer):

$$\mathit{type}(r, \texttt{reference}) = \mathit{typeOf}(r)$$
$$\mathit{type}(\_, t) = t$$

$$\begin{array}{l}
\mathit{typeOf}(r) = \\
\quad \textbf{if } (r = \mathit{null}) \textbf{ then } \texttt{Null} \\
\quad \textbf{else case } \mathit{heap}(r) \textbf{ of} \\
\qquad\qquad \mathit{Object}(c, \mathit{fields}) \rightarrow \textbf{case } \mathit{initState}(r) \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathit{New}(pc) \rightarrow (c, pc)_{new} \\
\qquad\qquad\qquad\qquad \mathit{InInit} \rightarrow \mathit{InInit} \\
\qquad\qquad\qquad\qquad \mathit{Complete} \rightarrow c
\end{array}$$

Checking JVM$_{\mathcal{O}}$ instructions

$$
\begin{array}{l}
check_O(meth)(instr, maxOpd, pc, regT, opdT) = \\
\quad check_C(meth)(instr, maxOpd, pc, regT, opdT) \wedge \\
\quad endinit(meth, instr, regT) \vee \\
\quad \textbf{case } instr \textbf{ of} \\
\qquad New(c) \rightarrow \neg over(maxOpd, opdT, 1) \\
\qquad GetField(t, c/f) \rightarrow opdT \sqsubseteq_{\text{suf}} c \wedge \\
\qquad\qquad\qquad\qquad \neg over(maxOpd, opdT, size(t) - 1) \\
\qquad PutField(t, c/f) \rightarrow opdT \sqsubseteq_{\text{suf}} c \cdot t \\
\qquad InstanceOf(c) \rightarrow opdT \sqsubseteq_{\text{suf}} \texttt{Object} \\
\qquad Checkcast(c) \rightarrow opdT \sqsubseteq_{\text{suf}} \texttt{Object}
\end{array}
$$

Checking JVM$_{\mathcal{O}}$ instructions (continued)

$$
\begin{array}{l}
check_{\mathcal{O}}(meth)(instr, maxOpd, pc, regT, opdT) = \\
\quad \textbf{case } instr \textbf{ of} \\
\quad\quad InvokeSpecial(\_, c/m) \rightarrow \\
\quad\quad\quad \textbf{let } [c'] \cdot \_ = take(opdT, 1 + argSize(c/m)) \\
\quad\quad\quad length(opdT) > argSize(c/m) \wedge \\
\quad\quad\quad opdT \sqsubseteq_{\text{suf}} argTypes(c/m) \wedge \\
\quad\quad\quad \neg over(maxOpd, opdT, retSize(c/m) - argSize(c/m) - 1) \wedge \\
\quad\quad\quad \textbf{if } methNm(m) = \texttt{"<init>"} \textbf{ then} \\
\quad\quad\quad\quad initCompatible(meth, c', c) \\
\quad\quad\quad \textbf{else } c' \sqsubseteq c \\
\quad\quad InvokeVirtual(\_, c/m) \rightarrow \\
\quad\quad\quad opdT \sqsubseteq_{\text{suf}} c \cdot argTypes(c/m) \wedge \\
\quad\quad\quad \neg over(maxOpd, opdT, retSize(c/m) - argSize(c/m) - 1)
\end{array}
$$

Checking JVM$_{\mathcal{O}}$ instructions (continued)

Check: When a constructor (not in class `Object`) returns, it has invoked a constructor either in the same class or in the superclass.

$$
\begin{array}{l}
endinit(c/m, instr, regT) = \\
\quad \textbf{if } instr = Return(\_) \wedge methNm(m) = \texttt{"<init>"} \wedge c \neq \texttt{Object} \\
\quad \textbf{then} \\
\qquad 0 \in dom(regT) \wedge regT(0) \neq InInit \\
\quad \textbf{else } True
\end{array}
$$

Constraint: An `<init>` method does not use $Store(\_, 0)$.

Check: If an object is un-initialized, then the invoked constructor is in the class of the object. If an object is partially initialized, then the invoked constructor is in the same class as the invoking constructor or in the direct superclass.

$$
\begin{array}{l}
initCompatible(\_, (c, \_)_{new}, c') \;\; = (c = c') \\
initCompatible(c/m, InInit, c') = (c = c' \vee super(c) = c')
\end{array}
$$

Checking JVM$_{\mathcal{E}}$ instructions

$\mathit{WordType} = \ldots \mid \texttt{retAddr}(\mathit{Pc})$
$\mathit{VerifyType} = \ldots \mid \texttt{retAddr}(\mathit{Pc})$

$$defensiveVM_E = defensiveScheme_C(check_E, trustfulVM_E)$$

$$\begin{array}{l}
check_E(meth)(instr, maxOpd, pc, regT, opdT) = \\
\quad check_O(meth)(instr, maxOpd, pc, regT, opdT) \vee \\
\quad \textbf{case } instr \textbf{ of} \\
\qquad Store(\texttt{addr}, x) \rightarrow length(opdT) > 0 \wedge isRetAddr(top(opdT)) \\
\qquad Athrow \rightarrow opdT \sqsubseteq_{\text{suf}} \texttt{Throwable} \\
\qquad Jsr(o) \rightarrow \neg overflow(maxOpd, opdT, 1) \\
\qquad Ret(x) \rightarrow isRetAddr(regT(x))
\end{array}$$

$$\begin{array}{ll}
isRetAddr(\texttt{retAddr}(_)) & = True \\
isRetAddr(_) & = False
\end{array}$$

Checks are monotonic

Definition of $\sqsubseteq_{\text{reg}}$:

$$regS \sqsubseteq_{\text{reg}} regT = \forall\, x \in dom(regT):\\ x \in dom(regS) \wedge regS(x) \sqsubseteq regT(x)$$

Definition of $\sqsubseteq_{\text{seq}}$:

$$xs \sqsubseteq_{\text{seq}} ys = (length(xs) = length(ys)) \wedge\\ \forall\, i < length(xs) : xs(i) \sqsubseteq ys(i)$$

Abbreviation: We write $check(meth, pc, regT, opdT)$ for $check_E(meth)(code(meth)(pc), maxOpd(meth), pc, regT, opdT)$.

Lemma. If $regS \sqsubseteq_{\text{reg}} regT$, $opdS \sqsubseteq_{\text{seq}} opdT$ and $check(meth, pc, regT, opdT) = True$, then $check(meth, pc, regS, opdS) = True$.