Java and JVM in a Nutshell

Robert Stärk, Joachim Schmid, Egon Börger

April 4, 2003

The main rules and functions which constitute the ASM models of Java and the JVM. When refering to these rules please cite the Jbook and not this document. This document is not for distribution. The Home-Page of Jbook is

http://www.inf.ethz.ch/~jbook/

where this document and more information about Jbook is available.

1 Java rules

$$
\begin{aligned}
execJava = \; & execJava_I \\
& execJava_C \\
& execJava_O \\
& execJava_E \\
& execJava_T
\end{aligned}
$$

$execJava_I =$
$\quad execJavaExp_I$
$\quad execJavaStm_I$

$execJava_C =$
$\quad execJavaExp_C$
$\quad execJavaStm_C$

$execJava_O =$
$\quad execJavaExp_O$

$execJava_E =$
$\quad execJavaExp_E$
$\quad execJavaStm_E$

$$
\begin{array}{l}
execJava_T = \\
\quad execJavaStm_T
\end{array}
$$

$$
\begin{array}{l}
context(pos) = \textbf{if } pos = firstPos \lor restbody/pos \in Bstm \cup Exp \textbf{ then} \\
\qquad\qquad\qquad restbody/pos \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad restbody/up(pos)
\end{array}
$$

$$
\begin{array}{l}
yieldUp(result) = \\
\quad restbody := restbody[result/up(pos)] \\
\quad pos \quad\quad := up(pos)
\end{array}
$$

$$
\begin{array}{l}
yield(result) = \\
\quad restbody := restbody[result/pos]
\end{array}
$$

$$
\begin{array}{l}
execJavaExp_I = \textbf{case } context(pos) \textbf{ of} \\
\quad lit \to yield(JLS(lit)) \\
\\
\quad loc \to yield(locals(loc)) \\
\\
\quad uop\, {}^{\alpha} exp \to pos := \alpha \\
\quad uop\, {}^{\blacktriangleright} val \to yieldUp(JLS(uop, val)) \\
\\
\quad {}^{\alpha} exp_1 \; bop\, {}^{\beta} exp_2 \to pos := \alpha \\
\quad {}^{\blacktriangleright} val \; bop\, {}^{\beta} exp \quad \to pos := \beta \\
\quad {}^{\alpha} val_1 \; bop\, {}^{\blacktriangleright} val_2 \;\; \to \textbf{if } \neg(bop \in divMod \land isZero(val_2)) \textbf{ then} \\
\qquad\qquad\qquad\qquad\quad yieldUp(JLS(bop, val_1, val_2)) \\
\\
\quad loc = {}^{\alpha} exp \to pos := \alpha \\
\quad loc = {}^{\blacktriangleright} val \to locals := locals \oplus \{(loc, val)\} \\
\qquad\qquad\qquad\quad yieldUp(val) \\
\\
\quad {}^{\alpha} exp_0 \,?\, {}^{\beta} exp_1 \;:\; {}^{\gamma} exp_2 \to pos := \alpha \\
\quad {}^{\blacktriangleright} val \,?\, {}^{\beta} exp_1 \;:\; {}^{\gamma} exp_2 \;\; \to \textbf{if } val \textbf{ then } pos := \beta \textbf{ else } pos := \gamma \\
\quad {}^{\alpha} True \,?\, {}^{\blacktriangleright} val \;:\; {}^{\gamma} exp \quad \to yieldUp(val) \\
\quad {}^{\alpha} False \,?\, {}^{\beta} exp \;:\; {}^{\blacktriangleright} val \;\; \to yieldUp(val)
\end{array}
$$

$$
\begin{array}{l}
\mathit{execJavaStm}_I = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of}\\
\quad ; \rightarrow \mathit{yield}(\mathit{Norm})\\
\quad {}^{\alpha}\mathit{exp}; \rightarrow \mathit{pos} := \alpha\\
\quad {}^{\blacktriangleright}\mathit{val}; \rightarrow \mathit{yieldUp}(\mathit{Norm})\\
\\
\quad \texttt{break}\ \mathit{lab}; \rightarrow \mathit{yield}(\mathit{Break}(\mathit{lab}))\\
\quad \texttt{continue}\ \mathit{lab}; \rightarrow \mathit{yield}(\mathit{Continue}(\mathit{lab}))\\
\quad \mathit{lab} : {}^{\alpha}\mathit{stm} \rightarrow \mathit{pos} := \alpha\\
\quad \mathit{lab} : {}^{\blacktriangleright}\mathit{Norm} \rightarrow \mathit{yieldUp}(\mathit{Norm})\\
\quad \mathit{lab} : {}^{\blacktriangleright}\mathit{Break}(\mathit{lab}_b) \rightarrow \textbf{if } \mathit{lab} = \mathit{lab}_b \textbf{ then } \mathit{yieldUp}(\mathit{Norm})\\
\qquad\qquad \textbf{else } \mathit{yieldUp}(\mathit{Break}(\mathit{lab}_b))\\
\quad \mathit{lab} : {}^{\blacktriangleright}\mathit{Continue}(\mathit{lab}_c) \rightarrow \textbf{if } \mathit{lab} = \mathit{lab}_c \textbf{ then } \mathit{yield}(\mathit{body}/\mathit{pos})\\
\qquad\qquad \textbf{else } \mathit{yieldUp}(\mathit{Continue}(\mathit{lab}_c))\\
\quad \mathit{phrase}({}^{\blacktriangleright}\mathit{abr}) \rightarrow \textbf{if } \mathit{pos} \neq \mathit{firstPos} \wedge \mathit{propagatesAbr}(\mathit{restbody}/\mathit{up}(\mathit{pos})) \textbf{ then}\\
\qquad\qquad \mathit{yieldUp}(\mathit{abr})\\
\\
\quad \{\,\} \rightarrow \mathit{yield}(\mathit{Norm})\\
\quad \{{}^{\alpha_1}\mathit{stm}_1 \ldots {}^{\alpha_n}\mathit{stm}_n\} \rightarrow \mathit{pos} := \alpha_1\\
\quad \{{}^{\alpha_1}\mathit{Norm} \ldots {}^{\blacktriangleright}\mathit{Norm}\} \rightarrow \mathit{yieldUp}(\mathit{Norm})\\
\quad \{{}^{\alpha_1}\mathit{Norm} \ldots {}^{\blacktriangleright}\mathit{Norm}^{\alpha_{i+1}}\mathit{stm}_{i+1} \ldots {}^{\alpha_n}\mathit{stm}_n\} \rightarrow \mathit{pos} := \alpha_{i+1}\\
\\
\quad \texttt{if}\ ({}^{\alpha}\mathit{exp})\,{}^{\beta}\mathit{stm}_1\ \texttt{else}\ {}^{\gamma}\mathit{stm}_2 \rightarrow \mathit{pos} := \alpha\\
\quad \texttt{if}\ ({}^{\blacktriangleright}\mathit{val})\,{}^{\beta}\mathit{stm}_1\ \texttt{else}\ {}^{\gamma}\mathit{stm}_2 \rightarrow \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \beta \textbf{ else } \mathit{pos} := \gamma\\
\quad \texttt{if}\ ({}^{\alpha}\mathit{True})\,{}^{\blacktriangleright}\mathit{Norm}\ \texttt{else}\ {}^{\gamma}\mathit{stm} \rightarrow \mathit{yieldUp}(\mathit{Norm})\\
\quad \texttt{if}\ ({}^{\alpha}\mathit{False})\,{}^{\beta}\mathit{stm}\ \texttt{else}\ {}^{\blacktriangleright}\mathit{Norm} \rightarrow \mathit{yieldUp}(\mathit{Norm})\\
\\
\quad \texttt{while}\ ({}^{\alpha}\mathit{exp})\,{}^{\beta}\mathit{stm} \rightarrow \mathit{pos} := \alpha\\
\quad \texttt{while}\ ({}^{\blacktriangleright}\mathit{val})\,{}^{\beta}\mathit{stm} \rightarrow \textbf{if } \mathit{val} \textbf{ then } \mathit{pos} := \beta \textbf{ else } \mathit{yieldUp}(\mathit{Norm})\\
\quad \texttt{while}\ ({}^{\alpha}\mathit{True})\,{}^{\blacktriangleright}\mathit{Norm} \rightarrow \mathit{yieldUp}(\mathit{body}/\mathit{up}(\mathit{pos}))\\
\\
\quad \mathit{Type}\ x; \rightarrow \mathit{yield}(\mathit{Norm})
\end{array}
$$

$$
\begin{array}{l}
\mathit{execJavaExp}_C = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of}\\
\quad c.f \rightarrow \textbf{if } \mathit{initialized}(c) \textbf{ then } \mathit{yield}(\mathit{globals}(c/f)) \textbf{ else } \mathit{initialize}(c)\\
\quad c.f = {}^{\alpha}\mathit{exp} \rightarrow \mathit{pos} := \alpha\\
\quad c.f = {}^{\blacktriangleright}\mathit{val} \rightarrow \textbf{if } \mathit{initialized}(c) \textbf{ then}\\
\qquad\qquad \mathit{globals}(c/f) := \mathit{val}\\
\qquad\qquad \mathit{yieldUp}(\mathit{val})\\
\qquad \textbf{else } \mathit{initialize}(c)\\
\\
\quad c.m^{\alpha}(\mathit{exps}) \rightarrow \mathit{pos} := \alpha\\
\quad c.m^{\blacktriangleright}(\mathit{vals}) \rightarrow \textbf{if } \mathit{initialized}(c) \textbf{ then } \mathit{invokeMethod}(\mathit{up}(\mathit{pos}), c/m, \mathit{vals})\\
\qquad \textbf{else } \mathit{initialize}(c)\\
\\
\quad (\,) \rightarrow \mathit{yield}([\,])\\
\quad ({}^{\alpha_1}\mathit{exp}_1, \ldots, {}^{\alpha_n}\mathit{exp}_n) \rightarrow \mathit{pos} := \alpha_1\\
\quad ({}^{\alpha_1}\mathit{val}_1, \ldots, {}^{\blacktriangleright}\mathit{val}_n) \rightarrow \mathit{yieldUp}([\mathit{val}_1, \ldots, \mathit{val}_n])\\
\quad ({}^{\alpha_1}\mathit{val}_1, \ldots, {}^{\blacktriangleright}\mathit{val}_i, {}^{\alpha_{i+1}}\mathit{exp}_{i+1} \ldots {}^{\alpha_n}\mathit{exp}_n) \rightarrow \mathit{pos} := \alpha_{i+1}
\end{array}
$$

$initialize(c) =$
  if $classState(c) = Linked$ **then**
    $classState(c) := InProgress$
    forall $f \in staticFields(c)$
      $globals(f) := defaultVal(type(f))$
    $invokeMethod(pos, c/\texttt{<clinit>}, [\,])$
  if $classState(c) = Linked$ **then**
    $initWait(c) := \emptyset$
    $initThread(c) := thread$
  if $classState(c) = InProgress \wedge initThread(c) \neq thread$ **then**
    $exec(thread) := Waiting$
    $cont(thread) := (frames, (meth, restbody, pos, locals))$
    $initWait(c) := initWait(c) \cup \{thread\}$
  if $classState(c) = Unusable$ **then**
    $fail(\texttt{NoClassDefFoundErr})$

$execJavaStm_C =$ **case** $context(pos)$ **of**
  $\texttt{static}\ {}^{\alpha}stm \rightarrow$ **let** $c = classNm(meth)$
    if $c = \texttt{Object} \vee initialized(super(c))$ **then** $pos := \alpha$
    else $initialize(super(c))$
  $\texttt{static}\ {}^{\alpha}Return \rightarrow yieldUp(Return)$

  $\texttt{return}\ {}^{\alpha}exp; \rightarrow pos := \alpha$
  $\texttt{return}\ {}^{\blacktriangleright}val; \rightarrow yieldUp(Return(val))$
  $\texttt{return}; \rightarrow yield(Return)$
  $lab : {}^{\blacktriangleright}Return \rightarrow yieldUp(Return)$
  $lab : {}^{\blacktriangleright}Return(val) \rightarrow yieldUp(Return(val))$
  $Return \rightarrow$ **if** $pos = firstPos \wedge \neg null(frames)$ **then**
      $exitMethod(Norm)$
  $Return(val) \rightarrow$ **if** $pos = firstPos \wedge \neg null(frames)$ **then**
      $exitMethod(val)$

  ${}^{\blacktriangleright}Norm; \rightarrow yieldUp(Norm)$

$invokeMethod(nextPos, c/m, values)$
  $\mid Native \in modifiers(c/m) =$
    $invokeNative(c/m, values)$
  $\mid$ **otherwise** $=$
    $frames := push(frames, (meth, restbody, nextPos, locals))$
    $meth := c/m$
    $restbody := body(c/m)$
    $pos := firstPos$
    $locals := zip(argNames(c/m), values)$

```
exitMethod(result) =
  let (oldMeth, oldPgm, oldPos, oldLocals) = top(frames)
  meth   := oldMeth
  pos    := oldPos
  locals := oldLocals
  frames := pop(frames)
  if methNm(meth) = "<clinit>" ∧ result = Norm then
    restbody                    := oldPgm
    classState(classNm(meth)) := Initialized
  elseif methNm(meth) = "<init>" ∧ result = Norm then
    restbody := oldPgm[locals("this")/oldPos]
  else
    restbody := oldPgm[result/oldPos]


execJavaExp_O = case context(pos) of
  this → yield(locals("this"))

  new c → if initialized(c) then create ref
              heap(ref) := Object(c, {(f, defaultVal(type(f)))
                                       | f ∈ instanceFields(c)})
              waitSet(ref) := ∅
              locks(ref)   := 0
              if c ⪯_h Thread then
                exec(ref) := NotStarted
                sync(ref) := []
                interruptedFlag(ref) := False
              yield(ref)
          else initialize(c)

  ^α exp.c/f → pos := α
  ▸ref.c/f → if ref ≠ null then yieldUp(getField(ref, c/f))

  ^α exp_1.c/f = ^β exp_2 → pos := α
  ▸ref.c/f = ^β exp    → pos := β
  ^α ref.c/f = ▸val    → if ref ≠ null then
    setField(ref, c/f, val)
    yieldUp(val)

  ^α exp instanceof c → pos := α
  ▸ref instanceof c → yieldUp(ref ≠ null ∧ classOf(ref) ⪯ c)

  (c)^α exp → pos := α
  (c)▸ref → if ref = null ∨ classOf(ref) ⪯ c then yieldUp(ref)

  ^α exp.c/m^β(exps) → pos := α
  ▸ref.c/m^β(exps) → pos := β
  ^α ref.c/m▸(vals) → if ref ≠ null then
    let c′ = case callKind(up(pos)) of
                Virtual → lookup(classOf(ref), c/m)
                Super   → lookup(super(classNm(meth)), c/m)
                Special → c
    invokeMethod(up(pos), c′/m, [ref] · vals)
```

$$
\begin{array}{ll}
\mathit{failUp}(\mathit{exc}) & = \mathit{yieldUp}(\texttt{throw}\,\texttt{new}\ \mathit{exc}();) \\
\mathit{fail}(\mathit{exc}) & = \mathit{yield}(\texttt{throw}\,\texttt{new}\ \mathit{exc}();)
\end{array}
$$

$$
\begin{array}{l}
\mathit{execJavaStm}_E = \textbf{case}\ \mathit{context}(\mathit{pos})\ \textbf{of} \\
\quad \texttt{throw}\ {}^{\alpha}\mathit{exp}; \rightarrow \mathit{pos} := \alpha \\
\quad \texttt{throw}\ {}^{\blacktriangleright}\mathit{ref}; \rightarrow \textbf{if}\ \mathit{ref} = \mathit{null}\ \ \textbf{then}\ \mathit{failUp}(\texttt{NullPointerException}) \\
\qquad\qquad\qquad\quad \textbf{else}\ \mathit{yieldUp}(\mathit{Exc}(\mathit{ref})) \\
\\
\quad \texttt{try}\ {}^{\alpha}\mathit{stm}\ \texttt{catch}\ \ldots \quad \rightarrow \mathit{pos} := \alpha \\
\quad \texttt{try}\ {}^{\blacktriangleright}\mathit{Norm}\ \texttt{catch}\ \ldots \ \rightarrow \mathit{yieldUp}(\mathit{Norm}) \\
\quad \texttt{try}\ {}^{\blacktriangleright}\mathit{Exc}(\mathit{ref})\ \texttt{catch}\ (c_1\ x_1)\ {}^{\beta_1}\mathit{stm}_1 \ldots \texttt{catch}\ (c_n\ x_n)\ {}^{\beta_n}\mathit{stm}_n \ \rightarrow \\
\qquad \textbf{if}\ \exists\, 1 \le j \le n : \mathit{classOf}(\mathit{ref}) \preceq_{\mathrm{h}}\ c_j\ \textbf{then} \\
\qquad\quad \textbf{let}\ j = \min\{i \mid \mathit{classOf}(\mathit{ref}) \preceq_{\mathrm{h}}\ c_i\} \\
\qquad\quad \mathit{pos} \quad := \beta_j \\
\qquad\quad \mathit{locals} := \mathit{locals} \oplus \{(x_j, \mathit{ref})\} \\
\qquad \textbf{else}\ \mathit{yieldUp}(\mathit{Exc}(\mathit{ref})) \\
\quad \texttt{try}\ {}^{\blacktriangleright}\mathit{abr}\ \texttt{catch}\ (c_1\ x_1)\ {}^{\beta_1}\mathit{stm}_1 \ldots \texttt{catch}\ (c_n\ x_n)\ {}^{\beta_n}\mathit{stm}_n \ \rightarrow \mathit{yieldUp}(\mathit{abr}) \\
\quad \texttt{try}\ {}^{\alpha}\mathit{Exc}(\mathit{ref}) \ldots \texttt{catch}\ (c_i\ x_i)\ {}^{\blacktriangleright}\mathit{Norm} \ldots \ \rightarrow \mathit{yieldUp}(\mathit{Norm}) \\
\quad \texttt{try}\ {}^{\alpha}\mathit{Exc}(\mathit{ref}) \ldots \texttt{catch}\ (c_i\ x_i)\ {}^{\blacktriangleright}\mathit{abr} \ldots \quad \rightarrow \mathit{yieldUp}(\mathit{abr}) \\
\\
\quad {}^{\alpha}\mathit{stm}_1\ \texttt{finally}\ {}^{\beta}\mathit{stm}_2 \ \rightarrow \mathit{pos} := \alpha \\
\quad {}^{\blacktriangleright}\mathit{Norm}\ \texttt{finally}\ {}^{\beta}\mathit{stm} \rightarrow \mathit{pos} := \beta \\
\quad {}^{\blacktriangleright}\mathit{abr}\ \texttt{finally}\ {}^{\beta}\mathit{stm} \quad \rightarrow \mathit{pos} := \beta \\
\quad {}^{\alpha}s\ \texttt{finally}\ {}^{\blacktriangleright}\mathit{Norm} \quad \rightarrow \mathit{yieldUp}(s) \\
\quad {}^{\alpha}s\ \texttt{finally}\ {}^{\blacktriangleright}\mathit{abr} \qquad \rightarrow \mathit{yieldUp}(\mathit{abr}) \\
\\
\quad \mathit{lab} :\ {}^{\blacktriangleright}\mathit{Exc}(\mathit{ref}) \quad \rightarrow \mathit{yieldUp}(\mathit{Exc}(\mathit{ref})) \\
\quad \texttt{static}\ {}^{\alpha}\mathit{Exc}(\mathit{ref}) \rightarrow \\
\qquad \textbf{if}\ \mathit{classOf}(\mathit{ref}) \preceq_{\mathrm{h}} \texttt{Error}\ \textbf{then} \\
\qquad\quad \mathit{yieldUp}(\mathit{Exc}(\mathit{ref})) \\
\qquad \textbf{else} \\
\qquad\quad \mathit{failUp}(\texttt{ExceptionInInitializerError}) \\
\quad \mathit{Exc}(\mathit{ref}) \rightarrow \textbf{if}\ \mathit{pos} = \mathit{firstPos} \wedge \neg \mathit{null}(\mathit{frames})\ \textbf{then} \\
\qquad\qquad\qquad \mathit{exitMethod}(\mathit{Exc}(\mathit{ref})) \\
\qquad\qquad\qquad \textbf{if}\ \mathit{methNm}(\mathit{meth}) = \texttt{"<clinit>"}\ \textbf{then} \\
\qquad\qquad\qquad\quad \mathit{classState}(\mathit{classNm}(\mathit{meth})) := \mathit{Unusable}
\end{array}
$$

$$
\begin{array}{l}
\mathit{execJavaExp}_E = \textbf{case}\ \mathit{context}(\mathit{pos})\ \textbf{of} \\
\quad {}^{\alpha}\mathit{val}_1\ \mathit{bop}\ {}^{\blacktriangleright}\mathit{val}_2 \quad \rightarrow \textbf{if}\ \mathit{bop} \in \mathit{divMod} \wedge \mathit{isZero}(\mathit{val}_2)\ \textbf{then} \\
\qquad\qquad\qquad\qquad \mathit{failUp}(\texttt{ArithmeticException}) \\
\quad {}^{\blacktriangleright}\mathit{ref}.c/f \qquad\quad \rightarrow \textbf{if}\ \mathit{ref} = \mathit{null}\ \ \textbf{then}\ \mathit{failUp}(\texttt{NullPointerException}) \\
\quad {}^{\alpha}\mathit{ref}.c/f = {}^{\blacktriangleright}\mathit{val} \ \rightarrow \textbf{if}\ \mathit{ref} = \mathit{null}\ \ \textbf{then}\ \mathit{failUp}(\texttt{NullPointerException}) \\
\quad {}^{\alpha}\mathit{ref}.c/m^{\blacktriangleright}(\mathit{vals}) \rightarrow \textbf{if}\ \mathit{ref} = \mathit{null}\ \ \textbf{then}\ \mathit{failUp}(\texttt{NullPointerException}) \\
\quad (c)^{\blacktriangleright}\mathit{ref} \qquad\quad \rightarrow \textbf{if}\ \mathit{ref} \neq \mathit{null} \wedge \mathit{classOf}(\mathit{ref}) \not\preceq c\ \ \textbf{then} \\
\qquad\qquad\qquad\qquad \mathit{failUp}(\texttt{ClassCastException})
\end{array}
$$

$$
\begin{array}{l}
\mathit{releaseLock}(\mathit{phrase}) = \\
\quad \textbf{let}\ [p] \cdot \mathit{rest} = \mathit{sync}(\mathit{thread}) \\
\quad \mathit{sync}(\mathit{thread}) := \mathit{rest} \\
\quad \mathit{locks}(p) \qquad := \mathit{locks}(p) - 1 \\
\quad \mathit{yieldUp}(\mathit{phrase})
\end{array}
$$

$killThread =$
$\quad waitSet(thread) := \emptyset$
$\quad exec(thread) \quad := Dead$
$\quad$ **forall** $q \in waitSet(thread)$
$\qquad exec(q) := Notified$

$execJavaStm_T =$ **case** $context(pos)$ **of**
$\quad$ `synchronized` $({}^{\alpha}exp)\,{}^{\beta}stm \to pos := \alpha$
$\quad$ `synchronized` $({}^{\blacktriangleright}ref)\,{}^{\beta}stm \to$
$\qquad$ **if** $ref = null$ **then** $failUp($`NullPointerException`$)$
$\qquad$ **else**
$\qquad\quad$ **if** $ref \in sync(thread)$ **then**
$\qquad\qquad sync(thread) := [ref] \cdot sync(thread)$
$\qquad\qquad locks(ref) \quad := locks(ref) + 1$
$\qquad\qquad pos \qquad := \beta$
$\qquad\quad$ **else**
$\qquad\qquad exec(thread) \quad := Synchronizing$
$\qquad\qquad syncObj(thread) := ref$
$\qquad\qquad cont(thread) \quad := (frames, (meth, restbody, \beta, locals))$
$\quad$ `synchronized` $({}^{\alpha}ref)\,{}^{\blacktriangleright}Norm \to releaseLock(Norm)$
$\quad$ `synchronized` $({}^{\alpha}ref)\,{}^{\blacktriangleright}abr \quad \to releaseLock(abr)$

$\quad$ `static` ${}^{\blacktriangleright}abr \to notifyThreadsWaitingForInitialization$
$\quad abr \to$ **if** $pos = firstPos \wedge null(frames)$ **then** $killThread$

$notifyThreadsWaitingForInitialization =$
$\quad$ **let** $c = classNm(meth)$
$\quad initWait(c) \quad := \emptyset$
$\quad initThread(c) := undef$
$\quad$ **forall** $q \in initWait(c)$
$\qquad exec(q) := Active$

$execJavaThread =$
$\quad$ **choose** $q \in dom(exec), runnable(q)$
$\qquad$ **if** $q = thread \wedge exec(q) = Active$ **then**
$\qquad\quad execJava$
$\qquad$ **else**
$\qquad\quad$ **if** $exec(thread) = Active$ **then**
$\qquad\qquad cont(thread) := (frames, (meth, restbody, pos, locals))$
$\qquad\quad thread := q$
$\qquad\quad run(q)$

$run(q) =$
$\quad switchCont(q)$
$\quad$ **if** $exec(q) = Synchronizing$ **then**
$\qquad synchronize(q)$
$\quad$ **if** $exec(q) = Notified$ **then**
$\qquad wakeup(q)$

$$
\begin{array}{l}
\mathit{switchCont}(q) = \\
\quad \textbf{let } (\mathit{frames}', (\mathit{meth}', \mathit{restbody}', \mathit{pos}', \mathit{locals}')) = \mathit{cont}(q) \\
\quad \begin{array}{ll}
\mathit{exec}(q) & := \mathit{Active} \\
\mathit{meth} & := \mathit{meth}' \\
\mathit{restbody} & := \mathit{restbody}' \\
\mathit{pos} & := \mathit{pos}' \\
\mathit{locals} & := \mathit{locals}' \\
\mathit{frames} & := \mathit{frames}'
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{synchronize}(q) = \\
\quad \begin{array}{ll}
\mathit{sync}(q) & := [\mathit{syncObj}(q)] \cdot \mathit{sync}(q) \\
\mathit{locks}(\mathit{syncObj}(q)) & := 1
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{wakeup}(q) = \\
\quad \mathit{locks}(\mathit{waitObj}(q)) := \mathit{occurrences}(\mathit{waitObj}(q), \mathit{sync}(q))
\end{array}
$$

$$
\begin{array}{l}
\mathit{invokeNative}(\mathit{meth}, \mathit{values}) \\
\quad \begin{array}{ll}
\mid \mathit{meth} = \texttt{Thread/start()} & = \mathit{start}(\mathit{values}(0)) \\
\mid \mathit{meth} = \texttt{Thread/interrupt()} & = \mathit{interrupt}(\mathit{values}(0)) \\
\mid \mathit{meth} = \texttt{Thread/interrupted()} & = \mathit{interrupted} \\
\mid \mathit{meth} = \texttt{Thread/isInterrupted()} & = \mathit{isInterrupted}(\mathit{values}(0)) \\
\mid \mathit{meth} = \texttt{Object/wait()} & = \mathit{wait}(\mathit{values}(0)) \\
\mid \mathit{meth} = \texttt{Object/notify()} & = \mathit{notify}(\mathit{values}(0)) \\
\mid \mathit{meth} = \texttt{Object/notifyAll()} & = \mathit{notifyAll}(\mathit{values}(0))
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{start}(\mathit{ref}) = \\
\quad \textbf{if } \mathit{exec}(\mathit{ref}) \neq \mathit{NotStarted} \textbf{ then} \\
\quad\quad \mathit{fail}(\texttt{IllegalThreadStateException}) \\
\quad \textbf{else} \\
\quad\quad \textbf{let } q = \mathit{getField}(\mathit{ref}, \texttt{Thread/"target"}) \\
\quad\quad\quad\quad \mathit{meth} = \mathit{lookup}(\mathit{classOf}(q), \texttt{Runnable/run()})/\texttt{run()} \\
\quad\quad \mathit{exec}(\mathit{ref}) := \mathit{Active} \\
\quad\quad \mathit{cont}(\mathit{ref}) := ([\,], (\mathit{meth}, \mathit{body}(\mathit{meth}), \mathit{firstPos}, \{(\texttt{"this"}, q)\})) \\
\quad\quad \mathit{yieldUp}(\mathit{Norm})
\end{array}
$$

$$
\begin{array}{l}
\mathit{interrupt}(q) = \\
\quad \mathit{yieldUp}(\mathit{Norm}) \\
\quad \textbf{if } \mathit{exec}(q) = \mathit{Waiting} \wedge \neg \mathit{classInitialization}(q) \textbf{ then} \\
\quad\quad \textbf{let } (\mathit{frames}', (\mathit{meth}', \mathit{restbody}', \mathit{pos}', \mathit{locals}')) = \mathit{cont}(q) \\
\quad\quad \textbf{let } \mathit{fail} = \mathit{restbody}'[\texttt{throw new InterruptedException}(); /\mathit{pos}'] \\
\quad\quad \textbf{let } \mathit{ref} = \mathit{waitObj}(q) \\
\quad\quad \begin{array}{ll}
\mathit{waitSet}(\mathit{ref}) & := \mathit{waitSet}(\mathit{ref}) \setminus \{q\} \\
\mathit{exec}(q) & := \mathit{Notified} \\
\mathit{cont}(q) & := (\mathit{frames}', (\mathit{meth}', \mathit{fail}, \mathit{pos}', \mathit{locals}')) \\
\mathit{interruptedFlag}(q) & := \mathit{False}
\end{array} \\
\quad \textbf{else} \\
\quad\quad \mathit{interruptedFlag}(q) := \mathit{True}
\end{array}
$$

$interrupted =$
  if $interruptedFlag(thread)$ **then**
    $interruptedFlag(thread) := False$
    $yield(True)$
  else
    $yield(False)$

$isInterrupted(q) =$
  if $interruptedFlag(q)$ **then**
    $yieldUp(True)$
  else
    $yieldUp(False)$

$wait(ref) =$
  if $ref \notin sync(thread)$ **then**
    $fail($`IllegalMonitorStateException`$)$
  else
    let $ret = restbody[Norm/up(pos)]$
    $waitSet(ref) := waitSet(ref) \cup \{thread\}$
    $locks(ref) := 0$
    $exec(thread) := Waiting$
    $waitObj(thread) := ref$
    $cont(thread) := (frames, (meth, ret, up(pos), locals))$
    $yieldUp(Norm)$

$notify(ref) =$
  if $ref \notin sync(thread)$ **then**
    $fail($`IllegalMonitorStateException`$)$
  else
    $yieldUp(Norm)$
    choose $q \in waitSet(ref)$
      $waitSet(ref) := waitSet(ref) \setminus \{q\}$
      $exec(q) := Notified$

$notifyAll(ref) =$
  if $ref \notin sync(thread)$ **then**
    $fail($`IllegalMonitorStateException`$)$
  else
    $waitSet(ref) := \emptyset$
    $yieldUp(Norm)$
    forall $q \in waitSet(ref)$
      $exec(q) := Notified$

2 JVM rules

Trustful execution

```
execVM_I(instr) =
  case instr of
    Prim(p)       → let (opd', ws) = split(opd, argSize(p))
                    if p ∈ divMod ⇒ sndArgIsNotZero(ws) then
                      opd := opd' · JVMS(p, ws)
                      pc  := pc + 1
    Dupx(s1, s2)  → let (opd', [ws1, ws2]) = splits(opd, [s1, s2])
                    opd := opd' · ws2 · ws1 · ws2
                    pc  := pc + 1
    Pop(s)        → let (opd', ws) = split(opd, s)
                    opd := opd'
                    pc  := pc + 1
    Load(t, x)    → if size(t) = 1 then opd := opd · [reg(x)]
                      else opd := opd · [reg(x), reg(x + 1)]
                    pc := pc + 1
    Store(t, x)   → let (opd', ws) = split(opd, size(t))
                    if size(t) = 1 then reg := reg ⊕ {(x, ws(0))}
                      else reg := reg ⊕ {(x, ws(0)), (x + 1, ws(1))}
                    opd := opd'
                    pc  := pc + 1
    Goto(o)       → pc := o
    Cond(p, o)    → let (opd', ws) = split(opd, argSize(p))
                    opd := opd'
                    if JVMS(p, ws) then pc := o else pc := pc + 1
    Halt          → halt := "Halt"
```

```
execVM_C(instr) =
  execVM_I(instr)
  case instr of
    GetStatic(_, c/f) → if initialized(c) then
                          opd := opd · globals(c/f)
                          pc  := pc + 1
                        else switch := InitClass(c)
    PutStatic(_, c/f) → if initialized(c) then
                          let (opd', ws) = split(opd, size(c/f))
                          globals(c/f) := ws
                          opd := opd'
                          pc  := pc + 1
                        else switch := InitClass(c)
    InvokeStatic(_, c/m) → if initialized(c) then
                             let (opd', ws) = split(opd, argSize(c/m))
                             opd    := opd'
                             switch := Call(c/m, ws)
                           else switch := InitClass(c)
    Return(t) → let (opd', ws) = split(opd, size(t))
                switch := Result(ws)
```

```
switchVM_C =
  case switch of
    Call(meth, args) → if ¬isAbstract(meth) then
                          pushFrame(meth, args)
                          switch := Noswitch
    Result(res)  → if implicitCall(meth) then popFrame(0, [])
                     else popFrame(1, res)
                   switch := Noswitch
    InitClass(c) → if classState(c) = Linked then
                     classState(c) := Initialized
                     forall f ∈ staticFields(c)
                       globals(c/f) := default(type(c/f))
                     pushFrame(c/<clinit>(), ())
                     if c = Object ∨ initialized(super(c)) then
                       switch := Noswitch
                     else
                       switch := InitClass(super(c))
```

```
pushFrame(newMeth, args) =
  stack := stack · [(pc, reg, opd, meth)]
  meth  := newMeth
  pc    := 0
  opd   := []
  reg   := makeRegs(args)
```

```
popFrame(offset, result) =
  let (stack′, [(pc′, reg′, opd′, meth′)]) = split(stack, 1)
  pc    := pc′ + offset
  reg   := reg′
  opd   := opd′ · result
  meth  := meth′
  stack := stack′
```

```
execVM_O(instr) =
  execVM_C(instr)
  case instr of
    New(c) →
      if initialized(c) then create r
        heap(r) := Object(c, {(f, defaultVal(f)) | f ∈ instanceFields(c)})
        opd := opd · [r]
        pc  := pc + 1
      else switch := InitClass(c)
    GetField(_, c/f) → let (opd', [r]) = split(opd, 1)
                       if r ≠ null then
                         opd := opd' · getField(r, c/f)
                         pc  := pc + 1
    PutField(_, c/f) → let (opd', [r] · ws) = split(opd, 1 + size(c/f))
                       if r ≠ null then
                         setField(r, c/f, ws)
                         pc  := pc + 1
                         opd := opd'
    InvokeSpecial(_, c/m) →
      let (opd', [r] · ws) = split(opd, 1 + argSize(c/m))
      if r ≠ null then
        opd    := opd'
        switch := Call(c/m, [r] · ws)
    InvokeVirtual(_, c/m) →
      let (opd', [r] · ws) = split(opd, 1 + argSize(c/m))
      if r ≠ null then
        opd    := opd'
        switch := Call(lookup(classOf(r), c/m), [r] · ws)

    InstanceOf(c) → let (opd', [r]) = split(opd, 1)
                    opd := opd' · (r ≠ null ∧ classOf(r) ⊑ c)
                    pc  := pc + 1
    Checkcast(c)  → let r = top(opd)
                    if r = null ∨ classOf(r) ⊑ c then
                      pc := pc + 1
```

```
switchVM_E =
  switchVM_C
  case switch of
    Call(meth, args) → if isAbstract(meth) then
                          raise("AbstractMethodError")
    InitClass(c) → if unusable(c) then
                      raise("NoClassDefFoundError")
    Throw(r) → if ¬escapes(meth, pc, classOf(r)) then
                  let exc = handler(meth, pc, classOf(r))
                  pc     := handle(exc)
                  opd    := [r]
                  switch := Noswitch
                else
                  if methNm(meth) = "<clinit>" then
                    if ¬(classOf(r) ⪯_h Error) then
                      raise("ExceptionInInitializerError")
                      pc := undef
                    else switch := ThrowInit(r)
                  else popFrame(0, [])
    ThrowInit(r) → let c = classNm(meth)
                    classState(c) := Unusable
                    popFrame(0, [])
                    if ¬superInit(top(stack), c) then
                      switch := Throw(r)

superInit((_, _, _, m), c) =
  methNm(m) = "<clinit>" ∧ super(classNm(m)) = c


execVM_E(instr) =
  execVM_O(instr)
  case instr of
    Athrow → let [r] = take(opd, 1)
              if r ≠ null then switch := Throw(r)
                else raise("NullPointerException")
    Jsr(s) → opd := opd · [pc + 1]
              pc  := s
    Ret(x) → pc := reg(x)
    Prim(p) → let ws = take(opd, argSize(p))
                if p ∈ divMod ∧ sndArgIsZero(ws) then
                  raise("ArithmeticException")
    GetField(_, c/f) → let [r] = take(opd, 1)
                          if r = null then raise("NullPointerException")
    PutField(_, c/f) → let [r] · ws = take(opd, 1 + size(c/f))
                          if r = null then raise("NullPointerException")
    InvokeSpecial(_, c/m) →
      let [r] · ws = take(opd, 1 + argSize(c/m))
      if r = null then raise("NullPointerException")
    InvokeVirtual(_, c/m) →
      let [r] · ws = take(opd, 1 + argSize(c/m))
      if r = null then raise("NullPointerException")
    Checkcast(c) → let r = top(opd)
                    if r ≠ null ∧ ¬(classOf(r) ⊑ c) then
                      raise("ClassCastException")
```

$execVM_N =$
  if $meth =$ `Object/equals` **then**
    $switch := Result(reg(0) = reg(1))$
  elseif $meth =$ `Object/clone` **then**
    let $r = reg(0)$
    if $classOf(r) \preceq_h$ `Cloneable` **then**
      create r'
        $heap(r') := heap(r)$
        $switch := Result(r')$
    else
      $raise($`"CloneNotSupportedException"`$)$

$prepareClass(c) =$
  forall $f \in staticFields(c)$
    $globals(c/f) := defaultVal(type(c/f))$

$trustfulVM_I = execVM_I(code(pc))$

$trustfulScheme_C(execVM, switchVM) =$
  if $switch = Noswitch$ **then**
    $execVM(code(pc))$
  else
    $switchVM$

$trustfulVM_C = trustfulScheme_C(execVM_C, switchVM_C)$

$trustfulVM_O = trustfulScheme_C(execVM_O, switchVM_C)$

$trustfulVM_E = trustfulScheme_C(execVM_E, switchVM_E)$

$trustfulScheme_N(nativeVM, switchVM) =$
  if $switch = Noswitch \wedge isNative(meth)$ **then**
    $nativeVM$
  else
    $trustfulScheme_C(execVM_E, switchVM)$

$trustfulVM_N = trustfulScheme_N(execVM_N, switchVM_E)$

Defensive execution

$pushFrame(c/m, args) =$
  $stack := stack \cdot [(pc, reg, opd, meth)]$
  $meth := c/m$
  $pc := 0$
  $opd := [\,]$
  $reg := makeRegs(args)$
  if $methNm(m) =$ `"<init>"` **then**
    let $[r] \cdot \_ = args$
    if $c =$ `Object` **then**
      $initState(r) := Complete$
    else
      $initState(r) := InInit$

$execVM_E(instr) =$
$\quad execVM_O(instr)$
$\quad$ **case** $instr$ **of**
$\quad\quad \ldots$
$\quad\quad Jsr(s) \rightarrow opd := opd \cdot [(pc + 1, \texttt{retAddr}(s))]$
$\quad\quad\quad\quad\quad\quad pc \;\; := s$
$\quad\quad \ldots$

$defensiveScheme_I(check, trustfulVM) =$
$\quad$ **if** $\neg validCodeIndex(code, pc) \vee$
$\quad\quad \neg check(code(pc), maxOpd, type(reg), type(opd))$ **then**
$\quad\quad\quad halt :=$ `"Runtime check failed"`
$\quad$ **else**
$\quad\quad trustfulVM$

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

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

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

$$defensiveVM_O = defensiveScheme_C(check_O, trustfulVM_O)$$

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

$defensiveScheme_N(check, trustfulVM) =$
$\quad$ **if** $isNative(meth)$ **then**
$\quad\quad$ **if** $check(meth)$ **then** $trustfulVM$
$\quad\quad$ **else** $halt :=$ `"unknown native method"`
$\quad$ **else**
$\quad\quad defensiveScheme_C(check_E, trustfulVM)$

$$defensiveVM_N = defensiveScheme_N(check_N, trustfulVM_N)$$

Diligent execution

$propagateVM_I(code, succ, pc) =$
$\quad$ **forall seq** $(s, regS, opdS) \in succ(code(pc), pc, regV_{pc}, opdV_{pc})$
$\quad\quad propagateSucc(code, s, regS, opdS)$

```
propagateSucc(code, s, regS, opdS) =
  if s ∉ dom(visited) then
    if validCodeIndex(code, s) then
      regV_s     := {(x, t) | (x, t) ∈ regS, validReg(t, s)}
      opdV_s     := [if validOpd(t, s) then t else unusable | t ∈ opdS]
      visited(s) := True
      changed(s) := True
    else
      halt := "Verification failed (invalid code index)"
  elseif regS ⊑_reg regV_s ∧ opdS ⊑_seq opdV_s then
    skip
  elseif length(opdS) = length(opdV_s) then
    regV_s     := regV_s ⊔_reg regS
    opdV_s     := opdV_s ⊔_opd opdS
    changed(s) := True
  else
    halt := "Propagate failed"
```

```
initVerify(meth) =
  visited(0) := True
  changed(0) := True
  regV_0     := formals(meth)
  opdV_0     := []
  forall i ∈ dom(visited), i ≠ 0
    visited(i) := undef
    changed(i) := undef
    regV_i     := undef
    opdV_i     := undef
  forall s ∈ dom(enterJsr)
    enterJsr(s) := ∅
  forall s ∈ dom(leaveJsr)
    leaveJsr(s) := ∅
```

```
switchVM_C =
  ...
  case switch of
    InitClass(c) → if classState(c) = Referenced then
                     linkClass(c)
```

```
linkClass(c) =
  let classes = {super(c)} ∪ implements(c)
  if c = Object ∨ ∀ c′ ∈ classes : classState(c′) ≥ Linked then
    prepareVerify(c)
  elseif ¬cyclicInheritance(c) then
    choose c′ ∈ classes, classState(c′) = Referenced
      linkClass(c′)
  else
    halt := "Cyclic Inheritance: " · classNm(c)
```

$prepareVerify(c) =$
if $constraintViolation(c)$ **then**
$halt := violationMsg(classNm(c))$
else
let $verifyMeths' = [(c/m) \mid m \in dom(methods(cEnv(c))), \neg null(code(c/m))]$
$verifyMeths := verifyMeths'$
$verifyClass := c$
$initVerify(top(verifyMeths'))$
$prepareClass(c)$

$propagateVM_E(code, succ, pc) =$
$propagateVM_I(code, succ, pc)$
case $code(pc)$ **of**
$Jsr(s) \rightarrow enterJsr(s) := \{pc\} \cup enterJsr(s)$
$\quad$ **forall seq** $(i, x) \in leaveJsr(s), i \notin dom(changed)$
if $regV_i(x) =$ `retAddr`(s) **then**
$propagateJsr(code, pc, s, i)$

$Ret(x) \rightarrow$ **let** `retAddr`$(s) = regV_{pc}(x)$
$\quad leaveJsr(s) := \{(pc, x)\} \cup leaveJsr(s)$
forall $j \in enterJsr(s), j \notin dom(changed)$
$propagateJsr(code, j, s, pc)$

$propagateJsr(code, j, s, i) =$
$propagateSucc(code, j+1, regJ \oplus mod(s) \lhd regV_i, opdV_i)$ **where**
$regJ = \{(x, t) \mid (x, t) \in mod(s) \unlhd regV_j, validJump(t, s) \wedge t \neq (\_, \_)_{new} \wedge t \neq InInit\}$

$diligentVM_I =$
if $dom(changed) \neq \emptyset$ **then**
$verifyScheme_I(code, maxOpd, propagateVM_I, succ_I, check_I)$
else
$trustfulVM_I$

$verifyScheme_I(code, maxOpd, propagateVM, succ, check) =$
choose $pc \in dom(changed)$
if $check(code(pc), maxOpd, regV_{pc}, opdV_{pc})$ **then**
$changed(pc) := undef$ **seq** $propagateVM(code, succ, pc)$
else
$halt :=$ `"Verification failed"`

$diligentScheme(verifyVM, execVM) =$
if $\neg isChecked$ **then**
$verifyVM$
else
$execVM$

$$
\begin{array}{l}
\mathit{diligentVM}_C = \mathit{diligentScheme}(\mathit{verifyVM}, \mathit{trustfulVM}_C)\\
\quad \textbf{where } \mathit{verifyVM} = \mathit{verifyScheme}_C(\mathit{propagateVM}_I, \mathit{succ}_C, \mathit{check}_C)
\end{array}
$$

$$
\begin{array}{l}
\mathit{verifyScheme}_C(\mathit{propagateVM}, \mathit{succ}, \mathit{check}) =\\
\quad \textbf{if } \mathit{dom}(\mathit{changed}) \neq \emptyset \textbf{ then}\\
\quad\quad \mathit{verifyScheme}_I(\mathit{code}(\mathit{meth}_v), \mathit{maxOpd}(\mathit{meth}_v), \mathit{propagateVM},\\
\quad\quad\quad\quad\quad\quad \mathit{succ}(\mathit{meth}_v), \mathit{check}(\mathit{meth}_v))\\
\quad \textbf{else}\\
\quad\quad \textbf{let } \mathit{verifyMeths}' = \mathit{drop}(\mathit{verifyMeths}, 1)\\
\quad\quad \mathit{verifyMeths} := \mathit{verifyMeths}'\\
\quad\quad \textbf{if } \mathit{length}(\mathit{verifyMeths}') > 0 \textbf{ then}\\
\quad\quad\quad \mathit{initVerify}(\mathit{top}(\mathit{verifyMeths}'))\\
\quad\quad \textbf{else}\\
\quad\quad\quad \mathit{classState}(\mathit{verifyClass}) := \mathit{Linked}
\end{array}
$$

$$
\begin{array}{l}
\mathit{diligentVM}_O = \mathit{diligentScheme}(\mathit{verifyVM}, \mathit{trustfulVM}_O)\\
\quad \textbf{where } \mathit{verifyVM} = \mathit{verifyScheme}_C(\mathit{propagateVM}_I, \mathit{succ}_O, \mathit{check}_O)
\end{array}
$$

$$
\begin{array}{l}
\mathit{diligentVM}_E = \mathit{diligentScheme}(\mathit{verifyVM}, \mathit{trustfulVM}_E)\\
\quad \textbf{where } \mathit{verifyVM} = \mathit{verifyScheme}_C(\mathit{propagateVM}_E, \mathit{succ}_E, \mathit{check}_E)
\end{array}
$$

$$
\begin{array}{l}
\mathit{verifyScheme}_N(\mathit{check}) =\\
\quad \textbf{if } \mathit{changed}(0) \wedge \mathit{isNative}(\mathit{meth}_v) \textbf{ then}\\
\quad\quad \textbf{if } \mathit{check}(\mathit{meth}_v) \textbf{ then}\\
\quad\quad\quad \mathit{changed}(0) := \mathit{undef}\\
\quad\quad \textbf{else}\\
\quad\quad\quad \mathit{halt} := \texttt{"Verification failed"}\\
\quad \textbf{else}\\
\quad\quad \mathit{verifyScheme}_C(\mathit{propagateVM}_E, \mathit{succ}_E, \mathit{check}_E)
\end{array}
$$

$$
\begin{array}{l}
\mathit{diligentVM}_N = \mathit{diligentScheme}(\mathit{verifyVM}, \mathit{trustfulVM}_N)\\
\quad \textbf{where } \mathit{verifyVM} = \mathit{verifyScheme}_N(\mathit{check}_N)
\end{array}
$$

Check functions

$$
\begin{array}{l}
\mathit{check}_I(\mathit{instr}, \mathit{maxOpd}, \mathit{regT}, \mathit{opdT}) =\\
\quad \textbf{case } \mathit{instr} \textbf{ of}\\
\quad\quad \mathit{Prim}(p) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \mathit{argTypes}(p) \wedge\\
\quad\quad\quad\quad \neg\mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{retSize}(p) - \mathit{argSize}(p))\\
\quad\quad \mathit{Dupx}(s_1, s_2) \rightarrow \textbf{let } [\mathit{ts}_1, \mathit{ts}_2] = \mathit{tops}(\mathit{opdT}, [s_1, s_2])\\
\quad\quad\quad\quad \mathit{length}(\mathit{opdT}) \geq s_1 + s_2 \wedge\\
\quad\quad\quad\quad \neg\mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, s_2) \wedge\\
\quad\quad\quad\quad \mathit{validTypeSeq}(\mathit{ts}_1) \wedge \mathit{validTypeSeq}(\mathit{ts}_2)\\
\quad\quad \mathit{Pop}(s) \rightarrow \mathit{length}(\mathit{opdT}) \geq s\\
\quad\quad \mathit{Load}(t, x) \rightarrow\\
\quad\quad\quad \textbf{if } \mathit{size}(t) = 1 \textbf{ then } [\mathit{regT}(x)] \sqsubseteq_{\mathrm{mv}} t \wedge \neg\mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, 1)\\
\quad\quad\quad \textbf{else } [\mathit{regT}(x), \mathit{regT}(x+1)] \sqsubseteq_{\mathrm{mv}} t \wedge \neg\mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, 2)\\
\quad\quad \mathit{Store}(t, \_) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} t\\
\quad\quad \mathit{Goto}(o) \rightarrow \mathit{True}\\
\quad\quad \mathit{Cond}(p, o) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \mathit{argTypes}(p)\\
\quad\quad \mathit{Halt} \rightarrow \mathit{True}
\end{array}
$$

$$
\begin{array}{l}
\mathit{check}_C(\mathit{meth})(\mathit{instr}, \mathit{maxOpd}, \mathit{regT}, \mathit{opdT}) = \\
\quad \mathit{check}_I(\mathit{instr}, \mathit{maxOpd}, \mathit{regT}, \mathit{opdT}) \vee \\
\quad \textbf{case } \mathit{instr} \textbf{ of} \\
\qquad \begin{array}{ll}
\mathit{GetStatic}(t, c/f) & \rightarrow \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{size}(t)) \\
\mathit{PutStatic}(t, c/f) & \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} t \\
\mathit{InvokeStatic}(t, c/m) & \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \mathit{argTypes}(c/m) \wedge \\
 & \quad \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{size}(t) - \\
 & \qquad\qquad\qquad \mathit{argSize}(c/m)) \\
\mathit{Return}(t) & \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \mathit{returnType}(\mathit{meth}) \wedge \\
 & \quad \mathit{returnType}(\mathit{meth}) \sqsubseteq_{\mathrm{mv}} t
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{check}_O(\mathit{meth})(\mathit{instr}, \mathit{maxOpd}, \mathit{regT}, \mathit{opdT}) = \\
\quad \mathit{check}_C(\mathit{meth})(\mathit{instr}, \mathit{maxOpd}, \mathit{regT}, \mathit{opdT}) \wedge \mathit{endinit}(\mathit{meth}, \mathit{instr}, \mathit{regT}) \vee \\
\quad \textbf{case } \mathit{instr} \textbf{ of} \\
\qquad \mathit{New}(c) \rightarrow \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, 1) \\
\qquad \mathit{GetField}(t, c/f) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} c \wedge \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{size}(t) - 1) \\
\qquad \mathit{PutField}(t, c/f) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} c \cdot t \\
\qquad \mathit{InvokeSpecial}(\_, c/m) \rightarrow \\
\qquad\quad \textbf{let } [c'] \cdot \_ = \mathit{take}(\mathit{opdT}, 1 + \mathit{argSize}(c/m)) \\
\qquad\quad \mathit{length}(\mathit{opdT}) > \mathit{argSize}(c/m) \wedge \\
\qquad\quad \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \mathit{argTypes}(c/m) \wedge \\
\qquad\quad \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{retSize}(c/m) - \mathit{argSize}(c/m) - 1) \wedge \\
\qquad\quad \textbf{if } \mathit{methNm}(m) = \texttt{"<init>"} \textbf{ then} \\
\qquad\qquad \mathit{initCompatible}(\mathit{meth}, c', c) \\
\qquad\quad \textbf{else } c' \sqsubseteq c \\
\qquad \mathit{InvokeVirtual}(\_, c/m) \rightarrow \\
\qquad\quad \mathit{opdT} \sqsubseteq_{\mathrm{suf}} c \cdot \mathit{argTypes}(c/m) \wedge \\
\qquad\quad \neg \mathit{overflow}(\mathit{maxOpd}, \mathit{opdT}, \mathit{retSize}(c/m) - \mathit{argSize}(c/m) - 1) \\
\qquad \mathit{InstanceOf}(c) \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \texttt{Object} \\
\qquad \mathit{Checkcast}(c) \quad \rightarrow \mathit{opdT} \sqsubseteq_{\mathrm{suf}} \texttt{Object}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{check}_N(c/m) = \\
\quad c/m = \texttt{Object}/\texttt{equals} \vee \\
\quad c/m = \texttt{Object}/\texttt{clone}
\end{array}
$$

Successor functions

$$
\begin{array}{l}
\mathit{succ}_I(\mathit{instr}, \mathit{pc}, \mathit{regT}, \mathit{opdT}) = \\
\quad \textbf{case } \mathit{instr} \textbf{ of} \\
\quad\quad \mathit{Prim}(p) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, \mathit{argSize}(p)) \cdot \mathit{returnType}(p))\} \\
\quad\quad \mathit{Dupx}(s_1, s_2) \rightarrow \\
\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, s_1+s_2) \cdot \\
\quad\quad\quad\quad\quad\quad \mathit{take}(\mathit{opdT}, s_2) \cdot \mathit{take}(\mathit{opdT}, s_1+s_2))\} \\
\quad\quad \mathit{Pop}(s) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, s))\} \\
\quad\quad \mathit{Load}(t, x) \rightarrow \\
\quad\quad\quad \textbf{if } \mathit{size}(t) = 1 \textbf{ then} \\
\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}, \mathit{opdT} \cdot [\mathit{regT}(x)])\} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}, \mathit{opdT} \cdot [\mathit{regT}(x), \mathit{regT}(x+1)])\} \\
\quad\quad \mathit{Store}(t, x) \rightarrow \\
\quad\quad\quad \textbf{if } \mathit{size}(t) = 1 \textbf{ then} \\
\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT} \oplus \{(x, \mathit{top}(\mathit{opdT}))\}, \mathit{drop}(\mathit{opdT}, 1))\} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT} \oplus \{(x, t_0), (x+1, t_1)\}, \mathit{drop}(\mathit{opdT}, 2))\} \\
\quad\quad\quad \textbf{where } [t_0, t_1] = \mathit{take}(\mathit{opdT}, 2) \\
\quad\quad \mathit{Goto}(o) \quad\;\; \rightarrow \{(o, \mathit{regT}, \mathit{opdT})\} \\
\quad\quad \mathit{Cond}(p, o) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, \mathit{argSize}(p))), \\
\quad\quad\quad\quad\quad\quad\quad\;\; (o, \mathit{regT}, \mathit{drop}(\mathit{opdT}, \mathit{argSize}(p)))\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{succ}_C(\mathit{meth})(\mathit{instr}, \mathit{pc}, \mathit{regT}, \mathit{opdT}) = \\
\quad \mathit{succ}_I(\mathit{instr}, \mathit{pc}, \mathit{regT}, \mathit{opdT}) \cup \\
\quad \textbf{case } \mathit{instr} \textbf{ of} \\
\quad\quad \mathit{GetStatic}(t, c/f) \quad\quad\; \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{opdT} \cdot t)\} \\
\quad\quad \mathit{PutStatic}(t, c/f) \quad\quad\; \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, \mathit{size}(t)))\} \\
\quad\quad \mathit{InvokeStatic}(t, c/m) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, \mathit{argSize}(c/m)) \cdot t)\} \\
\quad\quad \mathit{Return}(\mathit{mt}) \quad\quad\quad\quad \rightarrow \emptyset
\end{array}
$$

$$
\begin{array}{l}
\mathit{succ}_O(\mathit{meth})(\mathit{instr}, \mathit{pc}, \mathit{regT}, \mathit{opdT}) = \\
\quad \mathit{succ}_C(\mathit{meth})(\mathit{instr}, \mathit{pc}, \mathit{regT}, \mathit{opdT}) \cup \\
\quad \textbf{case } \mathit{instr} \textbf{ of} \\
\quad\quad \mathit{New}(c) \rightarrow \{(\mathit{pc}+1, \mathit{regS}, \mathit{opdS} \cdot [(c, \mathit{pc})_{\mathit{new}}])\} \\
\quad\quad\quad \textbf{where } \mathit{regS} \;= \{(x, t) \mid (x, t) \in \mathit{regT}, t \neq (c, \mathit{pc})_{\mathit{new}}\} \\
\quad\quad\quad\quad\quad\quad\;\; \mathit{opdS} = [\textbf{if } t = (c, \mathit{pc})_{\mathit{new}} \textbf{ then } \texttt{unusable} \textbf{ else } t \mid t \in \mathit{opdT}] \\
\quad\quad \mathit{GetField}(t, c/f) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, 1) \cdot t)\} \\
\quad\quad \mathit{PutField}(t, c/f) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, 1 + \mathit{size}(t)))\} \\
\quad\quad \mathit{InvokeSpecial}(t, c/m) \rightarrow \\
\quad\quad\quad \textbf{let } \mathit{opdT}' = \mathit{drop}(\mathit{opdT}, 1 + \mathit{argSize}(c/m)) \cdot t \\
\quad\quad\quad \textbf{if } \mathit{methNm}(m) = \texttt{"<init>"} \textbf{ then} \\
\quad\quad\quad\quad \textbf{case } \mathit{top}(\mathit{drop}(\mathit{opdT}, \mathit{argSize}(c/m))) \textbf{ of} \\
\quad\quad\quad\quad\quad (c, o)_{\mathit{new}} \rightarrow \{(\mathit{pc}+1, \mathit{regT}[c/(c, o)_{\mathit{new}}], \mathit{opdT}'[c/(c, o)_{\mathit{new}}])\} \\
\quad\quad\quad\quad\quad \mathit{InInit} \quad\;\; \rightarrow \textbf{let } c/\_ = \mathit{meth} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}[c/\mathit{InInit}], \mathit{opdT}'[c/\mathit{InInit}])\} \\
\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}, \mathit{opdT}')\} \\
\quad\quad \mathit{InvokeVirtual}(t, c/m) \rightarrow \\
\quad\quad\quad \textbf{let } \mathit{opdT}' = \mathit{drop}(\mathit{opdT}, 1 + \mathit{argSize}(c/m)) \cdot t \\
\quad\quad\quad \{(\mathit{pc}+1, \mathit{regT}, \mathit{opdT}')\} \\
\quad\quad \mathit{InstanceOf}(c) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, 1) \cdot [\texttt{int}])\} \\
\quad\quad \mathit{Checkcast}(t) \rightarrow \{(\mathit{pc}+1, \mathit{regT}, \mathit{drop}(\mathit{opdT}, 1) \cdot t)\}
\end{array}
$$

$$
\begin{array}{l}
succ_E(meth)(instr, pc, regT, opdT) = \\
\quad succ_O(meth)(instr, pc, regT, opdT) \cup allhandlers(instr, meth, pc, regT) \cup \\
\quad \textbf{case}\ instr\ \textbf{of} \\
\quad\quad Athrow \rightarrow \emptyset \\
\quad\quad Jsr(s) \rightarrow \{(s, regT, opdT \cdot [\texttt{retAddr}(s)])\} \\
\quad\quad Ret(x) \rightarrow \emptyset
\end{array}
$$

3 Compilation functions

$$
\begin{array}{ll}
\mathcal{E}(lit) & = Prim(lit) \\
\mathcal{E}(loc) & = Load(\mathcal{T}(loc), \overline{loc}) \\
\mathcal{E}(loc = exp) & = \mathcal{E}(exp) \cdot Dupx(0, size(\mathcal{T}(exp))) \cdot Store(\mathcal{T}(exp), \overline{loc}) \\
\mathcal{E}(\texttt{!}\ exp) & = \mathcal{B}_1(exp, \mathrm{una}_1) \cdot Prim(1) \cdot Goto(\mathrm{una}_2) \cdot \\
 & \quad \mathrm{una}_1 \cdot Prim(0) \cdot \mathrm{una}_2 \\
\mathcal{E}(uop\ exp) & = \mathcal{E}(exp) \cdot Prim(uop) \\
\mathcal{E}(exp_1\ bop\ exp_2) & = \mathcal{E}(exp_1) \cdot \mathcal{E}(exp_2) \cdot Prim(bop) \\
\mathcal{E}(exp_0\ ?\ exp_1\ :\ exp_2) & = \mathcal{B}_1(exp_0, \mathrm{if}_1) \cdot \mathcal{E}(exp_2) \cdot Goto(\mathrm{if}_2) \cdot \mathrm{if}_1 \cdot \mathcal{E}(exp_1) \cdot \mathrm{if}_2
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{S}(;) & = \epsilon \\
\mathcal{S}(exp;) & = \mathcal{E}(exp) \cdot Pop(size(\mathcal{T}(exp))) \\
\mathcal{S}(\{stm_1 \dots stm_n\}) & = \mathcal{S}(stm_1) \cdot \ldots \cdot \mathcal{S}(stm_n) \\
\mathcal{S}(\texttt{if}\ (exp)\ stm_1\ \texttt{else}\ stm_2) & = \mathcal{B}_1(exp, \mathrm{if}_1) \cdot \mathcal{S}(stm_2) \cdot Goto(\mathrm{if}_2) \cdot \\
 & \quad \mathrm{if}_1 \cdot \mathcal{S}(stm_1) \cdot \mathrm{if}_2 \\
\mathcal{S}(\texttt{while}\ (exp)\ stm) & = Goto(\mathrm{while}_1) \cdot \mathrm{while}_2 \cdot \mathcal{S}(stm) \cdot \\
 & \quad \mathrm{while}_1 \cdot \mathcal{B}_1(exp, \mathrm{while}_2) \\
\mathcal{S}(lab : stm) & = lab_c \cdot \mathcal{S}(stm) \cdot lab_b \\
\mathcal{S}(\texttt{continue}\ lab;) & = \textbf{let}\ [\mathrm{fin}_1, \dots, \mathrm{fin}_n] = finallyLabsUntil(lab) \\
 & \quad Jsr(\mathrm{fin}_1) \cdot \ldots \cdot Jsr(\mathrm{fin}_n) \cdot Goto(lab_c) \\
\mathcal{S}(\texttt{break}\ lab;) & = \textbf{let}\ [\mathrm{fin}_1, \dots, \mathrm{fin}_n] = finallyLabsUntil(lab) \\
 & \quad Jsr(\mathrm{fin}_1) \cdot \ldots \cdot Jsr(\mathrm{fin}_n) \cdot Goto(lab_b)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B}_1(\texttt{true}, lab) & = Goto(lab) \\
\mathcal{B}_1(\texttt{false}, lab) & = \epsilon \\
\mathcal{B}_1(\texttt{!}\ exp, lab) & = \mathcal{B}_0(exp, lab) \\
\mathcal{B}_1(exp_0\ ?\ exp_1\ :\ exp_2, lab) & = \mathcal{B}_1(exp_0, \mathrm{if}_1) \cdot \mathcal{B}_1(exp_2, lab) \cdot Goto(\mathrm{if}_2) \cdot \\
 & \quad \mathrm{if}_1 \cdot \mathcal{B}_1(exp_1, lab) \cdot \mathrm{if}_2 \\
\mathcal{B}_1(exp, lab) & = \mathcal{E}(exp) \cdot Cond(\texttt{ifne}, lab)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B}_0(\texttt{true}, lab) & = \epsilon \\
\mathcal{B}_0(\texttt{false}, lab) & = Goto(lab) \\
\mathcal{B}_0(\texttt{!}\ exp, lab) & = \mathcal{B}_1(exp, lab) \\
\mathcal{B}_0(exp_0\ ?\ exp_1\ :\ exp_2, lab) & = \mathcal{B}_1(exp_0, \mathrm{if}_1) \cdot \mathcal{B}_0(exp_2, lab) \cdot Goto(\mathrm{if}_2) \cdot \\
 & \quad \mathrm{if}_1 \cdot \mathcal{B}_0(exp_1, lab) \cdot \mathrm{if}_2 \\
\mathcal{B}_0(exp, lab) & = \mathcal{E}(exp) \cdot Cond(\texttt{ifeq}, lab)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}(c.f) & = GetStatic(\mathcal{T}(c/f), c/f) \\
\mathcal{E}(c.f = exp) & = \mathcal{E}(exp) \cdot Dupx(0, size(\mathcal{T}(exp))) \cdot PutStatic(\mathcal{T}(c/f), c/f) \\
\mathcal{E}(c.m(exps)) & = \mathcal{E}(exps) \cdot InvokeStatic(\mathcal{T}(c/m), c/m)
\end{array}
$$

$$
\mathcal{E}((exp_1, \dots, exp_n)) = \mathcal{E}(exp_1) \cdot \ldots \cdot \mathcal{E}(exp_n)
$$

$$
\begin{array}{l}
\mathcal{S}(\texttt{static}\ stm) = \mathcal{S}(stm)\\
\mathcal{S}(\texttt{return};) \quad = \textbf{let}\ [\mathrm{fin}_1,\ldots,\mathrm{fin}_n] = \mathit{finallyLabs}\\
\qquad\qquad \mathit{Jsr}(\mathrm{fin}_1)\cdot\ldots\cdot \mathit{Jsr}(\mathrm{fin}_n)\cdot \mathit{Return}(\texttt{void})\\
\mathcal{S}(\texttt{return}\ exp;) =\\
\quad \textbf{if}\ \mathit{finallyCodeToExec}\ \textbf{then}\\
\qquad \mathcal{E}(exp)\cdot \mathit{Store}(\mathcal{T}(exp),\overline{var})\cdot\\
\qquad \textbf{let}\ [\mathrm{fin}_1,\ldots,\mathrm{fin}_n] = \mathit{finallyLabs}\\
\qquad \mathit{Jsr}(\mathrm{fin}_1)\cdot\ldots\cdot \mathit{Jsr}(\mathrm{fin}_n)\cdot \mathit{Load}(\mathcal{T}(exp),\overline{var})\cdot \mathit{Return}(\mathcal{T}(exp))\\
\quad \textbf{else}\\
\qquad \mathcal{E}(exp)\cdot \mathit{Return}(\mathcal{T}(exp))
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}(\texttt{this}) & = \mathit{Load}(\texttt{addr},0)\\
\mathcal{E}(\texttt{new}\ c) & = \mathit{New}(c)\cdot \mathit{Dupx}(0,1)\\
\mathcal{E}(exp.c/f) & = \mathcal{E}(exp)\cdot \mathit{GetField}(\mathcal{T}(c/f),c/f)\\
\mathcal{E}(exp_1.c/f = exp_2) & = \mathcal{E}(exp_1)\cdot\mathcal{E}(exp_2)\cdot \mathit{Dupx}(1,\mathit{size}(\mathcal{T}(c/f)))\cdot\\
& \quad \mathit{PutField}(\mathcal{T}(c/f),c/f)\\
\mathcal{E}(exp.c/m(exps)) & = \mathcal{E}(exp)\cdot\mathcal{E}(exps)\cdot\\
& \quad \textbf{case}\ \mathit{callKind}(exp.c/m)\ \textbf{of}\\
& \qquad \mathit{Virtual} \to \mathit{InvokeVirtual}(\mathcal{T}(c/m),c/m)\\
& \qquad \mathit{Super} \quad \to \mathit{InvokeSpecial}(\mathcal{T}(c/m),c/m)\\
& \qquad \mathit{Special} \to \mathit{InvokeSpecial}(\mathcal{T}(c/m),c/m)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}(exp\ \texttt{instanceof}\ c) & = \mathcal{E}(exp)\cdot \mathit{InstanceOf}(c)\\
\mathcal{E}((c)exp) & = \mathcal{E}(exp)\cdot \mathit{Checkcast}(c)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}(\texttt{throw}\ exp;) = \mathcal{E}(exp)\cdot \mathit{Athrow}\\
\mathcal{S}(\texttt{try}\ stm\ \texttt{catch}\ (c_1\ x_1)\ stm_1 \ldots \texttt{catch}\ (c_n\ x_n)\ stm_n\ ) =\\
\quad \mathrm{try}\cdot\mathcal{S}(stm)\cdot \mathrm{tryEnd}\cdot \mathit{Goto}(\mathrm{end})\cdot\\
\quad \mathrm{handle}_1\cdot \mathit{Store}(\texttt{addr},\overline{x_1})\cdot\mathcal{S}(stm_1)\cdot \mathit{Goto}(\mathrm{end})\cdot\\
\quad \vdots\\
\quad \mathrm{handle}_n\cdot \mathit{Store}(\texttt{addr},\overline{x_n})\cdot\mathcal{S}(stm_n)\cdot \mathit{Goto}(\mathrm{end})\cdot\\
\quad \mathrm{end}\\
\mathcal{S}(stm_1\ \texttt{finally}\ stm_2) =\\
\quad \mathrm{try_f}\cdot\mathcal{S}(stm_1)\cdot \mathit{Jsr}(\mathrm{fin})\cdot \mathit{Goto}(\mathrm{end})\cdot\\
\quad \mathrm{default}\cdot \mathit{Store}(\texttt{addr},\overline{exc})\cdot \mathit{Jsr}(\mathrm{fin})\cdot \mathit{Load}(\texttt{addr},\overline{exc})\cdot \mathit{Athrow}\cdot\\
\quad \mathrm{fin}\cdot \mathit{Store}(\texttt{addr},\overline{ret})\cdot\mathcal{S}(stm_2)\cdot \mathit{Ret}(\overline{ret})\cdot\\
\quad \mathrm{end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{X}(\texttt{try}\ stm\ \texttt{catch}\ (c_1\ x_1)\ stm_1 \ldots \texttt{catch}\ (c_n\ x_n)\ stm_n\ ) =\\
\quad \mathcal{X}(stm)\cdot\\
\quad \mathcal{X}(stm_1)\cdot \mathit{Exc}(\mathrm{try},\mathrm{tryEnd},\mathrm{handle}_1,c_1)\cdot\\
\quad \vdots\\
\quad \mathcal{X}(stm_n)\cdot \mathit{Exc}(\mathrm{try},\mathrm{tryEnd},\mathrm{handle}_n,c_n)\\
\mathcal{X}(stm_1\ \texttt{finally}\ stm_2) =\\
\quad \mathcal{X}(stm_1)\cdot \mathit{Exc}(\mathrm{try_f},\mathrm{default},\mathrm{default},\texttt{Throwable})\cdot\mathcal{X}(stm_2)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{X}(\{stm_1\ldots stm_n\}) & = \mathcal{X}(stm_1)\cdot\ldots\cdot\mathcal{X}(stm_n)\\
\mathcal{X}(\texttt{if}\ (exp)\ stm_1\ \texttt{else}\ stm_2) & = \mathcal{X}(stm_1)\cdot\mathcal{X}(stm_2)\\
\mathcal{X}(\texttt{while}\ (exp)\ stm) & = \mathcal{X}(stm)\\
\mathcal{X}(lab:\ stm) & = \mathcal{X}(stm)\\
\mathcal{X}(\texttt{static}\ stm) & = \mathcal{X}(stm)\\
\mathcal{X}(\_) & = \epsilon
\end{array}
$$