Design for Reuse
via Structuring Techniques for ASMs

Case Study:

Decomposing and Layering the Java VM

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

Composition of ASMs via Standard Refinements

The challenge

Starting point: standard ASMs come with **parallel** execution of **atomic actions** in a **global state** providing

- strong foundational thesis
  Yuri Gurevich, ACM TCL 1(1), 2000
- clear notions of state & next-step-function

Goal: incorporate **non atomic structuring** concepts—SEQ, iteration, calling parameterized submachines, returning values, local state, error handling—**as standard refinements** to naturally support

- incremental and modular design of machines
- implementations leading to executable machines

ASMs with recursive parameterized submachines

$$\frac{}{[\![\textbf{skip}\,]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; \emptyset}$$

$$\frac{}{[\![f(t) := s]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; \{(f,a,b)\}} \qquad \text{if } a = [\![t]\!]^{\mathfrak{A}}_{\zeta} \text{ and } b = [\![s]\!]^{\mathfrak{A}}_{\zeta}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U \qquad [\![S]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; V}{[\![R\ S]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U \cup V}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U}{[\![\textbf{if } \varphi \textbf{ then } R \textbf{ else } S]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U} \qquad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}$$

$$\frac{[\![S]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U}{[\![\textbf{if } \varphi \textbf{ then } R \textbf{ else } S]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U} \qquad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \;\rhd\; U}{[\![\textbf{let } x = t \textbf{ in } R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U} \qquad \text{if } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

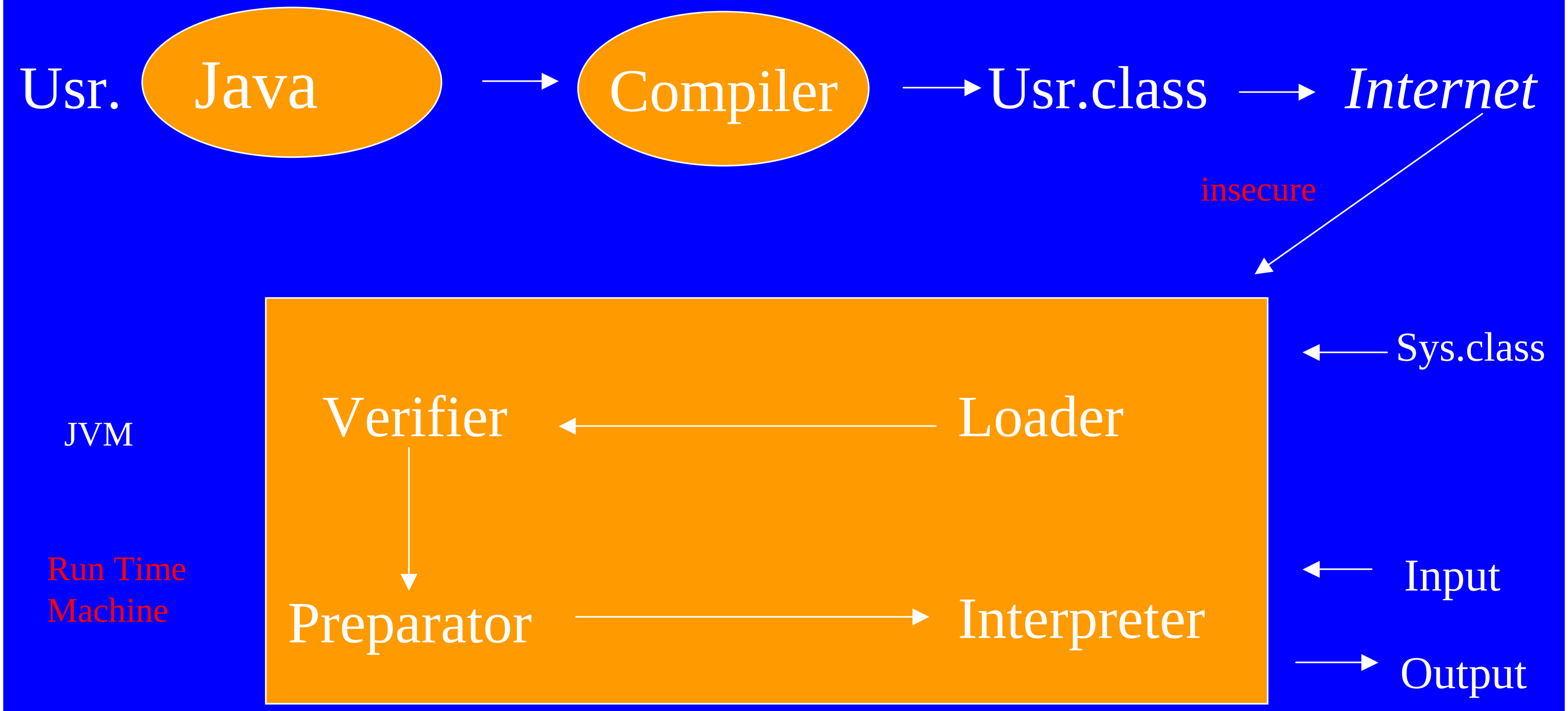
$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \;\rhd\; U_a \quad \text{for each } a \in I}{[\![\textbf{forall } x \textbf{ with } \varphi \textbf{ do } R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; \bigcup_{a \in I} U_a} \qquad \text{if } I = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True}\}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \;\rhd\; U}{[\![r(t)]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U} \qquad \begin{array}{l}\text{if } r(x) = R \text{ is a rule definition} \\ \text{and } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}\end{array}$$

E. Börger & J.Schmid: Composition and Submachine Concepts, LNCS 1862, 41-60 (2000)

The Problem

Java/JVM claimed by SUN to be a safe and secure, platform independent programming env for Internet: correctness problem for compiler, loader (name space support), verifier, access right checker (security manager) , interpreter.



Usr. Java → Compiler → Usr.class → Internet

insecure

Sys.class

JVM

Verifier ← Loader

Run Time Machine

Preparator → Interpreter

Input

Output

Goal of the ASM Java/JVM Project

Abstract (platform independent), rigorous but transparent, modular definition providing basis for mathematical and experimental analysis

- Reflecting SUN’s design decisions (faithful ground model)
- Offering correct high-level understanding (to be practically useful for programmers)
- Providing rigorous, implementation independent basis for
  - Analysis and Documentation (for designers) through
    - Mathematical verification
    - Experimental validation
    - Comparison of different implementations
  - Implementation (compiln, loading, bytecode verification, security schemes)

Main Result

A Structured and High-Level Definition of Java
and of its Provably Correct and Secure Implementation
on the Java Virtual Machine

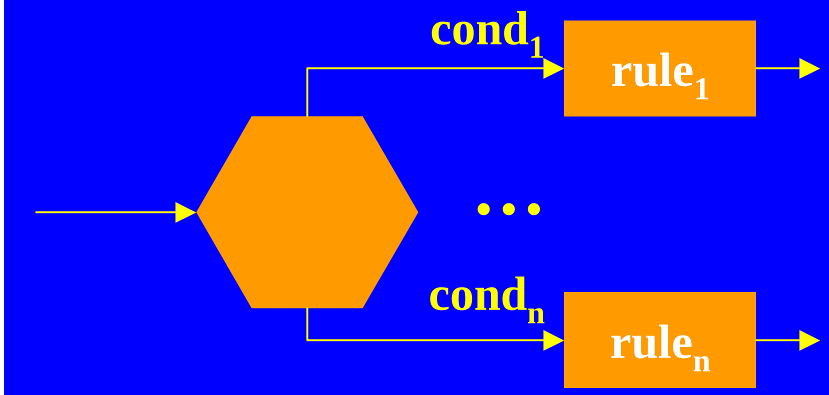
Theorem. Under explicitly stated conditions, any well-formed and well-typed Java program:

- upon correct compilation
- passes the verifier
- is executed on the JVM
- executes
  - without violating any run-time checks
  - correctly wrt Java source pgm semantics

Decomposition of JVM into Submachines

- trustfulVM: defines the execution functionality incrementally from language layered submachines execVM, switchVM
- defensiveVM: defines the verifier functionality, in terms of trustfulVM execution, from the language layered submachine check; calls trustfulVM for execution
- diligentVM: checks the constraints at link-time, using a language layered submachine verifyVM; calls trustfulVM for execution
- verifyVM built up from language layered submachines check, propagateVM, succ
- dynamicVM: refine execVM, switchVM by class loading/linking

Diagram notation for Control State ASMs



$cond_1$

$rule_1$

…

$cond_n$

$rule_n$

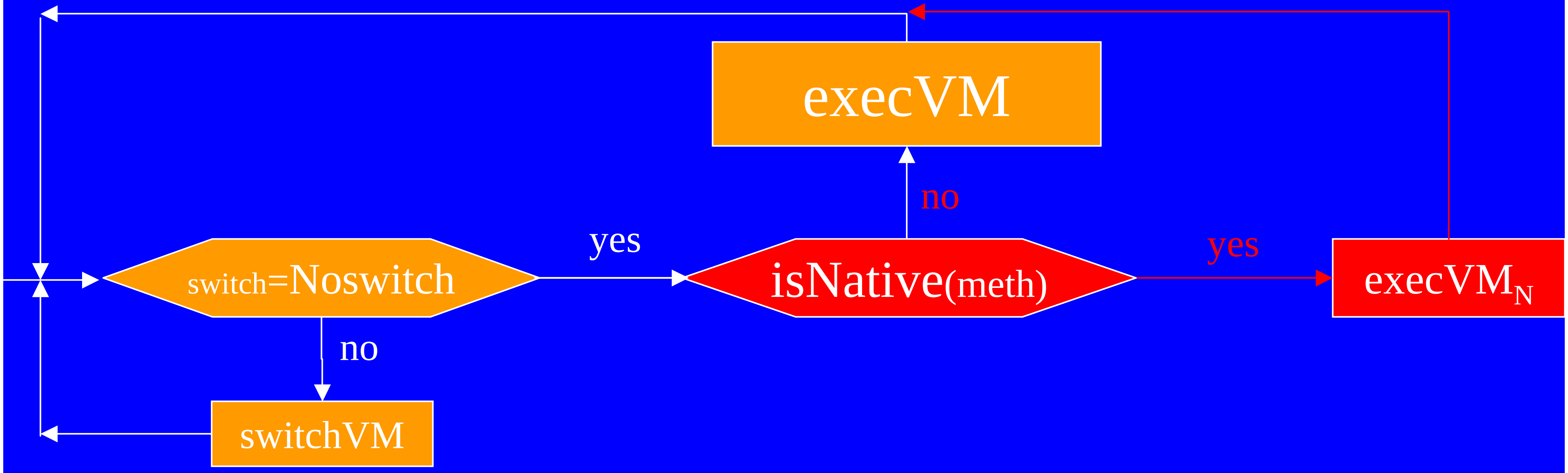
UML: combined
branching/action
nodes

meaning

```
if $ctl = i$ then
    if $cond_1$ then $rule_1$
                     $ctl := j_1$
    …
    if $cond_n$ then $rule_n$
                     $ctl := j_n$
```

labeling of the arrows
by "control" states
often suppressed

Stepwise refinement of trustfulVM



execVM and switchVM incrementally extended (language driven)

$\text{trustfulVM}_I = \text{execVM}_I \subseteq \text{execVM}_C \subseteq \text{execVM}_O \subseteq \text{execVM}_E$ instructionwise

defining changes of current frame

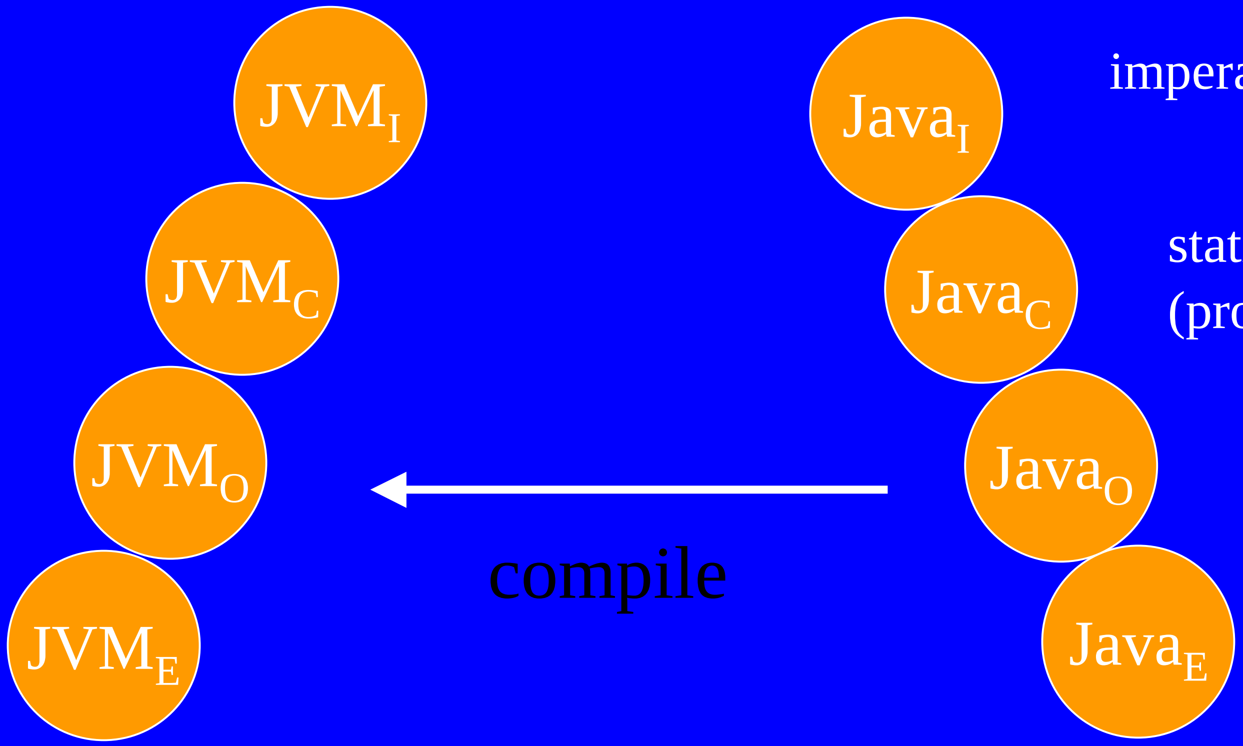
$\text{execVM}_N \subseteq \text{execVM}_D$

$\text{switchVM}_C \subseteq \text{switchVM}_E$ $\subseteq \text{switchVM}_D$ defining changes of frame stack

reflecting meth call/return, class initialization, capturing exceptions

and class loading/linking

Language driven layering of Java, JVM, compiler



JVM_I

JVM_C

JVM_O

JVM_E

compile

$Java_I$ imperative

$Java_C$ static class features (procedures)

$Java_O$ oo features

$Java_E$ exception handling

Split into horizontal language components (conservative extensions)

NB. Multiple Threads can be added in a conservative extension $Java_T$

Language driven decomposition of execVM & switchVM into parallel trustfulVM submachines

$\text{execVM} = \text{execVM}_I$	**imperative control constructs**
execVM_C	**static class features (modules)**
execVM_O	**oo features**
execVM_E	**exception handling**
$\text{execVM}_N \subseteq \text{execVM}_D$	**native JDK library meths (also for load/linking)**
$\text{switchVM} = \text{switchVM}_C$	method call/return & class initialization
switchVM_E	capturing exceptions
switchVM_D	loading and linking classes

NB. Grouping similar instructions into one parameterized abstract instr

(expanding type params a locally controllable data/operation refinement)

execVM_I: untyped 32-bit word oriented stack machine supporting exec o compiled while pgm instructions (e.g. purely imperative Java_I pgms)

STATE **frame**

code: Instr*

pc : Pc

reg: Reg → Word

(local variables)

opd: Word*

meth

Main guard (suppressed)

halt = undef

These 7 abstract instrs comprise already 150 out of 200 real JVM instructions

```
case $instr$ of
  $Prim(p)$         → let $(opd', ws) = split(opd, argSize(p))$
                      if $p \in divMod \Rightarrow sndArgIsNotZero(ws)$ then
                         $opd := opd' \cdot JVMS(p, ws)$
                         $pc \;\; := pc + 1$
  $Dupx(s_1, s_2)$  → let $(opd', [ws_1, ws_2]) = splits(opd, [s_1, s_2])$
                      $opd := opd' \cdot ws_2 \cdot ws_1 \cdot ws_2$
                      $pc \;\; := pc + 1$
  $Pop(s)$          → let $(opd', ws) = split(opd, s)$
                      $opd := opd'$
                      $pc \;\; := pc + 1$
  $Load(t, x)$      → if $size(t) = 1$ then $opd := opd \cdot [reg(x)]$
                         else $opd := opd \cdot [reg(x), reg(x + 1)]$
                      $pc := pc + 1$
  $Store(t, x)$     → let $(opd', ws) = split(opd, size(t))$
                      if $size(t) = 1$ then $reg := reg \oplus \{(x, ws(0))\}$
                         else $reg := reg \oplus \{(x, ws(0)), (x + 1, ws(1))$
                      $opd := opd'$
                      $pc \;\; := pc + 1$
  $Goto(o)$         → $pc := o$
  $Cond(p, o)$      → let $(opd', ws) = split(opd, argSize(p))$
                      $opd := opd'$
                      if $JVMS(p, ws)$ then $pc := o$ else $pc := pc +$
  $Halt$            → $halt :=$ "Halt"
```

Adding class variables, class initialization, class meth invocation & return

cEnv: Class ---> ClassFile providing name, kind, superclass, implemented interfaces, fields, meths,...

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

Frame stack manipulating submachine (push/pop)

```
$switchVM_C =$
  case $switch$ of
    $Call(meth, args) \rightarrow$ if $\neg isAbstract(meth)$ then
                                  $pushFrame(meth, args)$
                                  $switch := Noswitch$
    $Result(res) \rightarrow$ if $implicitCall(meth)$ then $popFrame(0, [])$
                          else $popFrame(1, res)$
                        $switch := Noswitch$
    $InitClass(c) \rightarrow$ if $classState(c) = Linked$ then
                          $classState(c) := Initialized$
                          forall $f \in staticFields(c)$
                            $globals(c/f) := default(type(c/f))$
                          $pushFrame(c/\texttt{<clinit>}())$
                          if $c = \texttt{Object} \vee initialized(super(c))$ the
                            $switch := Noswitch$
                          else
                            $switch := InitClass(super(c))$
```

Before its use, after having been loaded & linked (by dynamic VM), a class and its superclasses have to be initialized (implicit call of a clinit method upon exec of Put/Get/Invoke)

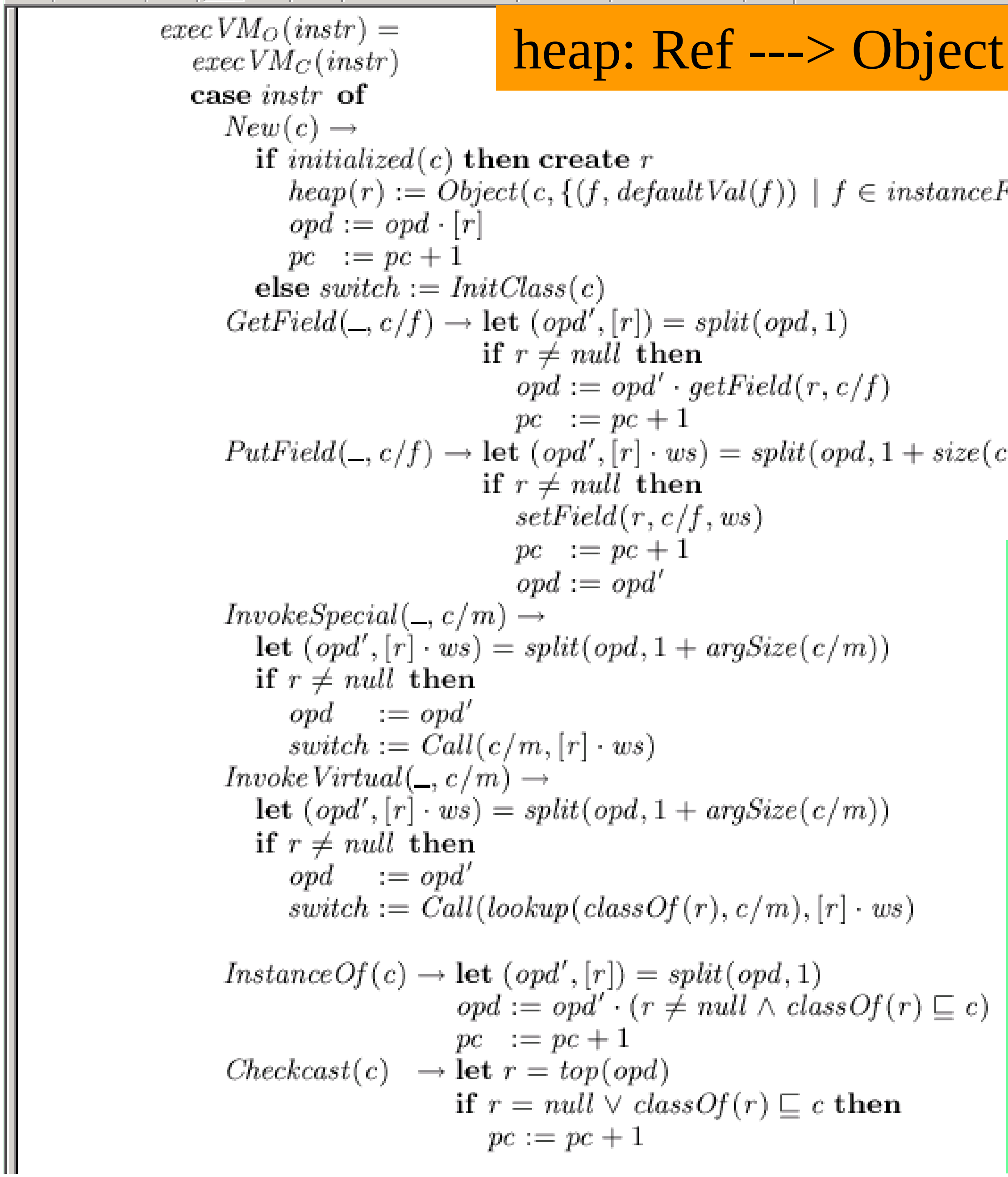
```
pushFrame(newMeth, args) =
stack := stack  [(pc, reg, opd, meth)
meth := newMeth
pc := 0
reg := makeRegs(args)
opd := [ ]
```

```
popFrame(offset; result) =
let (stack*; [(pc*; reg*; opd*; meth*)]) =
                                    split (stack; 1)
pc := pc* + offset
reg := reg*
opd := opd* . result
meth := meth*
stack := stack*
```

Instance creation/initializn, access, methods, type casts

heap: Ref ---> Object (Class, Map (Class/Field,Val))

Ref $\subseteq$ Word



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

    $InstanceOf(c) \rightarrow$ let $(opd', [r]) = split(opd, 1)$
                        $opd := opd' \cdot (r \neq null \wedge classOf(r) \sqsubseteq c)$
                        $pc \quad := pc + 1$
    $Checkcast(c) \quad \rightarrow$ let $r = top(opd)$
                        if $r = null \vee classOf(r) \sqsubseteq c$ then
                          $pc := pc + 1$
```

Instance method calls with
- early binding:InvokeSpecial, where the method reference contains the class of the implementing method
- late binding: Invoke Virtual, where the implementing method is looked up dynamically

execVM$_E$: adding the effect of exception handling instructions upon the current frame

```
$execVM_O(instr)$
case $instr$ of
    $Athrow \rightarrow$ let $[r] = take(opd, 1)$
                 if $r \neq null$ then $switch := Throw(r)$
                    else $raise($"NullPointerException"$)$
    $Jsr(s) \rightarrow opd := opd \cdot [pc + 1]$
                 $pc \quad := s$
    $Ret(x) \rightarrow pc := reg(x)$
    $Prim(p) \rightarrow$ let $ws = take(opd, argSize(p))$
                 if $p \in divMod \wedge sndArgIsZero(ws)$ then
                    $raise($"ArithmeticException"$)$
    $GetField(\_, c/f) \rightarrow$ let $[r] = take(opd, 1)$
                 if $r = null$ then $raise($"NullPointerException"$)$
    $PutField(\_, c/f) \rightarrow$ let $[r] \cdot ws = take(opd, 1 + size(c/f))$
                 if $r = null$ then $raise($"NullPointerException"$)$
    $InvokeSpecial(\_, c/m) \rightarrow$
        let $[r] \cdot ws = take(opd, 1 + argSize(c/m))$
        if $r = null$ then $raise($"NullPointerException"$)$
    $InvokeVirtual(\_, c/m) \rightarrow$
        let $[r] \cdot ws = take(opd, 1 + argSize(c/m))$
        if $r = null$ then $raise($"NullPointerException"$)$
    $Checkcast(c) \rightarrow$ let $r = top(opd)$
                 if $r \neq 0 \wedge \neg(classOf(r) \sqsubseteq c)$ then
                    $raise($"ClassCastException"$)$
```

Instructions to
- raise an exception
- jump to subroutine
- return from subroutine

Run-time exceptions

raise(c) defined e.g. by switch:=Call(fail(c), [ ]))

Adding frame stack manipulations for exceptions

Java try/catch implemented by tables of exceptions

(from, upto, handle, type)

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

$superInit((\_, \_, \_, m), c) =$
  $methNm(m) =$ "<clinit>" $\wedge super(classNm(m)) = c$
```

searching exc table of current method for handler

continue searching exc table of invoker

class becomes unusable when clinit exc not caught (recursively)

Specify Native Methods of JDK Libraries: 2 Exls

```
$execVM_N =$
  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"$)$
```

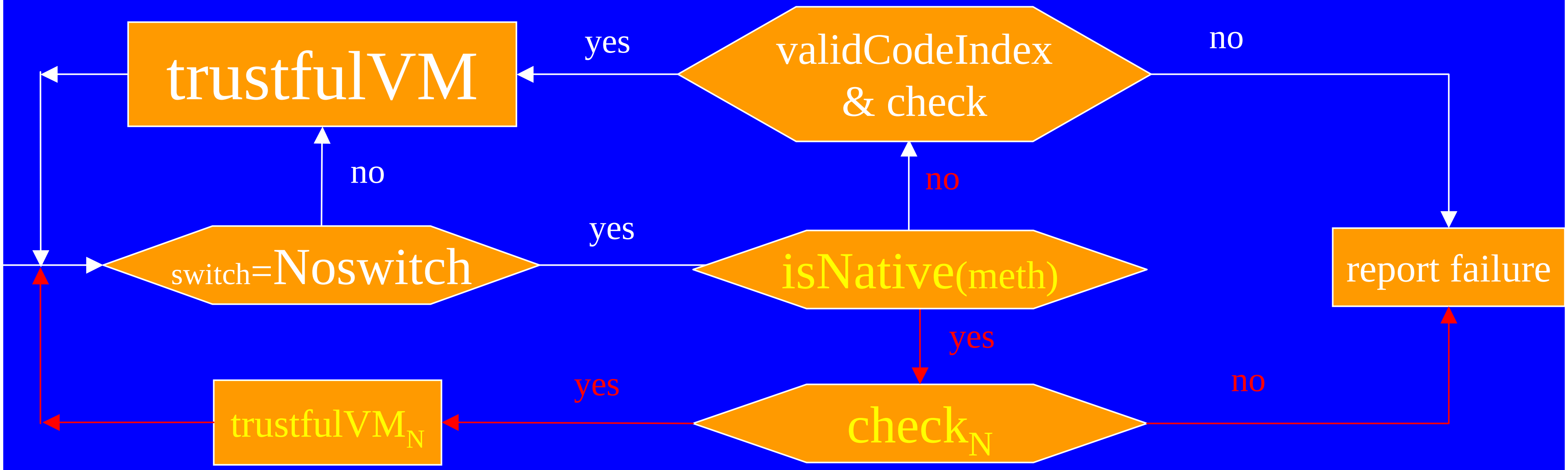
Executable version contains other native meths

e.g. for loading and resolving classes and for newInstance to create an instance for a given class object (see the extension $execVM_D$ of VM_N below)

Deriving the Bytecode Verifier Conditions from Type Checking Runtime Constraints

- Defensive VM: Checks at run-time, before every execution step, the “structural constraints” which describe the verifier functionality (restrictions on run-time data: argument types, valid Ret addresses, resource bounds,…) guaranteeing “safe” execution
- Static constraints (well-formedness) checked at link-time.
- Theorem: If Defensive VM executes P successfully, then so does Trustful VM, with the same semantical effect.

Stepwise refinement of defensiveVM



trustfulVM

yes

validCodeIndex & check

no

no

no

yes

switch=Noswitch

isNative(meth)

report failure

yes

yes

no

trustfulVM$_N$

check$_N$

check incrementally extended , language layered as for trustfulVM

i.e. check$_I$ extended by check$_C$
extended by check$_O$
extended by check$_E$
extended by check$_N$
extended by check$_D$

Lifting execVM to reg and opd types

Checking conditions formulated in terms of value types,
so that they can be lifted from run-time to link-time checks

Words/word fcts refined by type information, yielding (val,typ) pairs

JVM weakly typed: reg/opd locations can hold
int, float, low/high word of long or double

type frames (type(reg), type(opd)) where type selects types

Primops executed with right no/types of args, no opd over/underflow, double words not swapped/operated componentwise, locvars assigned when accessed

g. 15.2 Checking $\mathrm{JVM}_{\mathcal{I}}$ instructions

validTypeSeq([])
validTypeSeq([t]) = not isHigh(t)
validTypeSeq([ ]) = not isHigh(t)
isHigh(t) = (t=highLong or
t = highDouble)

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

[single] $\subseteq_{mv}$ single (for single = int,float)
[lowLD,highLD] $\subseteq_{mv}$ LD
for LD = Long,Double

$\subseteq_{mv}$ condition implies: a) reg(x) is assigned (regT(x)≠undef) when accessed
b) stored double words have correct low/high types

Checking JVM_C instructions for types of class fields and of method invocation arguments/results

a) types of values put into class fields are compatible with their declared types

b) types of actual args in class meth invocations are compatible with formal params

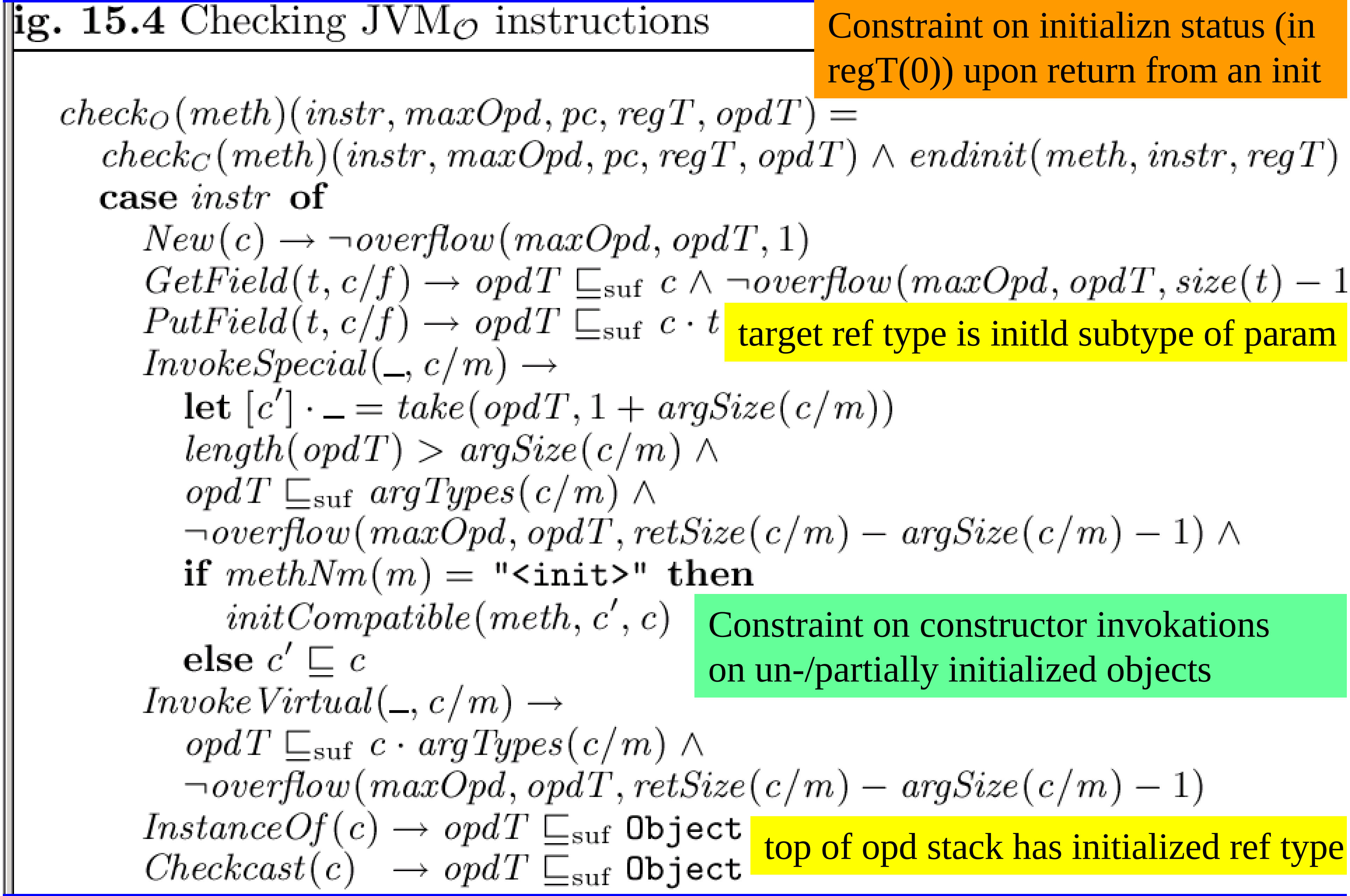
c) type of any returned result is compatible with the return type of the method, which in turn is compatible with the move type as specified by the instruction parameter

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

$[] \subseteq_{\mathrm{mv}}$ void

See later refinement by endinit for returns from instance initializn methods

Compatibility refined by inher hierchy, field access/method call only for initd instances



ig. 15.4 Checking JVM$_O$ instructions

Constraint on initializn status (in regT(0)) upon return from an init

```
$check_O(meth)(instr, maxOpd, pc, regT, opdT) =$
  $check_C(meth)(instr, maxOpd, pc, regT, opdT) \wedge endinit(meth, instr, regT)$
  case $instr$ of
    $New(c) \rightarrow \neg overflow(maxOpd, opdT, 1)$
    $GetField(t, c/f) \rightarrow opdT \sqsubseteq_{suf} c \wedge \neg overflow(maxOpd, opdT, size(t) - 1$
    $PutField(t, c/f) \rightarrow opdT \sqsubseteq_{suf} c \cdot t$
    $InvokeSpecial(\_, c/m) \rightarrow$
      let $[c'] \cdot \_ = take(opdT, 1 + argSize(c/m))$
      $length(opdT) > argSize(c/m) \wedge$
      $opdT \sqsubseteq_{suf} argTypes(c/m) \wedge$
      $\neg overflow(maxOpd, opdT, retSize(c/m) - argSize(c/m) - 1) \wedge$
      if $methNm(m) =$ "<init>" then
        $initCompatible(meth, c', c)$
      else $c' \sqsubseteq c$
    $InvokeVirtual(\_, c/m) \rightarrow$
      $opdT \sqsubseteq_{suf} c \cdot argTypes(c/m) \wedge$
      $\neg overflow(maxOpd, opdT, retSize(c/m) - argSize(c/m) - 1)$
    $InstanceOf(c) \rightarrow opdT \sqsubseteq_{suf}$ Object
    $Checkcast(c) \rightarrow opdT \sqsubseteq_{suf}$ Object
```

target ref type is initld subtype of param

Constraint on constructor invokations on un-/partially initialized objects

top of opd stack has initialized ref type

Compatibility refined by inher hierchy, field access/method call only for initd instances

Updating initState of objects in switchVM upon calling instance initialization meths along class hierarchy (only upon un-initialized or partially initialized objects)

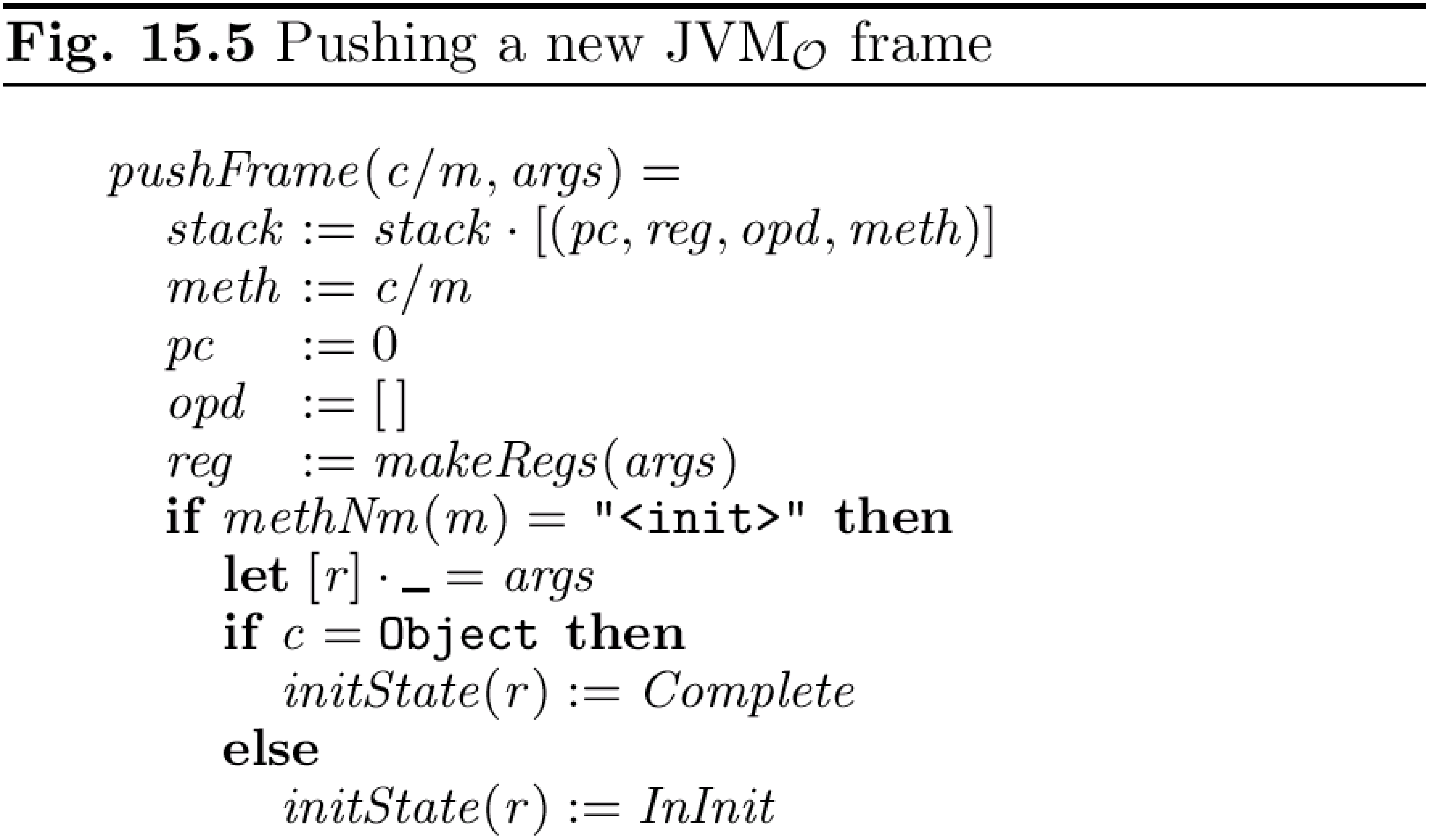


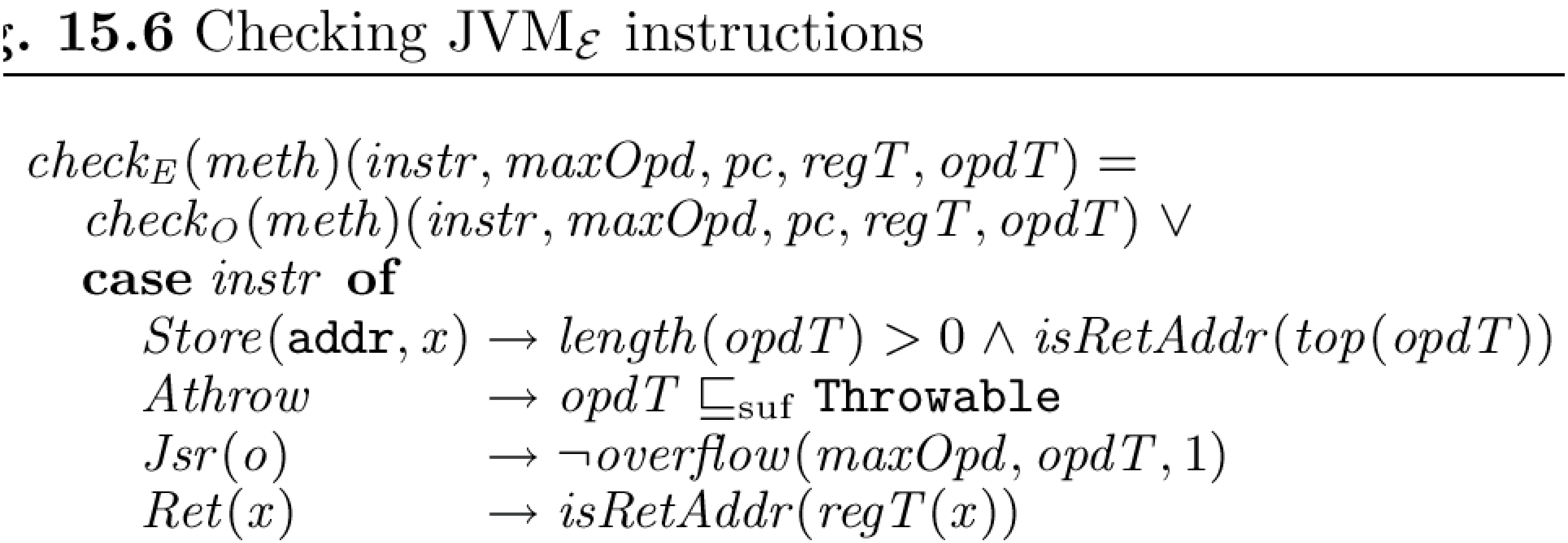
Fig. 15.5 Pushing a new JVM$_{\mathcal{O}}$ frame

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

A newly created object of class c is considered as un-initialized, reflected by setting initState(r):= New(pc) upon executing the instr New(c) in execVM$_O$

To guarantee: Athrow only applied upon throwable objects

Pgm counter values always denote valid addresses



. **15.6** Checking JVM$_{\mathcal{E}}$ instructions

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

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

In execVM$_E$ refine Jsr(s) to record that a retAddr is pushed on stack:

opd := opd · [(pc+1, retAddr(s))]

pc := s

No computed gotos: only Jsr generates retAddr & pushes them on stack

only Store can move a retAddr into a register

Checking native meths: 2 Exls

- Check guarantees that the VM has native code for the meth to execute upon its call
- Exls: equal and clone

check_N (c/m) =

c/m = Object/equals or c/m = Object/clone

- Implementation must assure that return val of native meths is of correct return type (bytecode verifier cannot check this, although it can be checked at run-time)

Bytecode Type Assignments

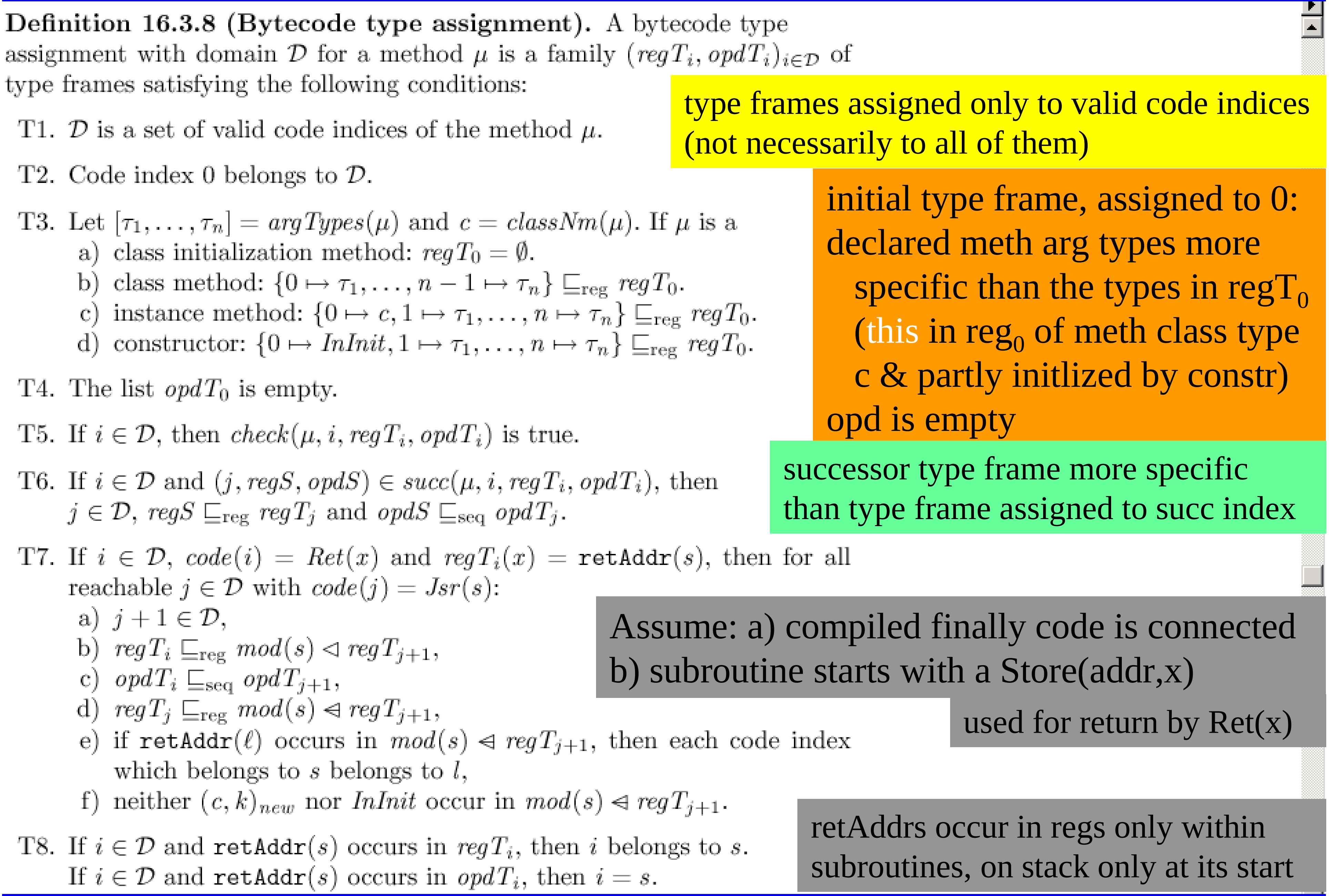
- Link-time verifiable type assignments (conditions) extracted from checking function of the Defensive VM

  Main problem: return addresses of Jsr(s), reached using Ret(x)
- Soundness Theorem: If P satisfies the type assignment conditions, then Defensive VM executes P without violating any run-time check.

  Proof by induction on runs of the Defensive VM
- Completeness Theorem: Bytecode generated by compile from a legal Java program does have type assignments.

  Inductive proof introduces certifying compiler assigning to each byte code instr also a type frame, which then can be shown to constitute a type assignment for the compiled code

Type assignments without subroutine call stacks



Definition 16.3.8 (Bytecode type assignment). A bytecode type assignment with domain $\mathcal{D}$ for a method μ is a family $(regT_i, opdT_i)_{i \in \mathcal{D}}$ of type frames satisfying the following conditions:

T1. $\mathcal{D}$ is a set of valid code indices of the method μ.

T2. Code index 0 belongs to $\mathcal{D}$.

T3. Let $[\tau_1, \ldots, \tau_n] = argTypes(\mu)$ and $c = classNm(\mu)$. If μ is a
  a) class initialization method: $regT_0 = \emptyset$.
  b) class method: $\{0 \mapsto \tau_1, \ldots, n-1 \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.
  c) instance method: $\{0 \mapsto c, 1 \mapsto \tau_1, \ldots, n \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.
  d) constructor: $\{0 \mapsto InInit, 1 \mapsto \tau_1, \ldots, n \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.

T4. The list $opdT_0$ is empty.

T5. If $i \in \mathcal{D}$, then $check(\mu, i, regT_i, opdT_i)$ is true.

T6. If $i \in \mathcal{D}$ and $(j, regS, opdS) \in succ(\mu, i, regT_i, opdT_i)$, then $j \in \mathcal{D}$, $regS \sqsubseteq_{\text{reg}} regT_j$ and $opdS \sqsubseteq_{\text{seq}} opdT_j$.

T7. If $i \in \mathcal{D}$, $code(i) = Ret(x)$ and $regT_i(x) = \texttt{retAddr}(s)$, then for all reachable $j \in \mathcal{D}$ with $code(j) = Jsr(s)$:
  a) $j + 1 \in \mathcal{D}$,
  b) $regT_i \sqsubseteq_{\text{reg}} mod(s) \lhd regT_{j+1}$,
  c) $opdT_i \sqsubseteq_{\text{seq}} opdT_{j+1}$,
  d) $regT_j \sqsubseteq_{\text{reg}} mod(s) \mathbin{⩤} regT_{j+1}$,
  e) if $\texttt{retAddr}(\ell)$ occurs in $mod(s) \mathbin{⩤} regT_{j+1}$, then each code index which belongs to s belongs to l,
  f) neither $(c, k)_{new}$ nor $InInit$ occur in $mod(s) \mathbin{⩤} regT_{j+1}$.

T8. If $i \in \mathcal{D}$ and $\texttt{retAddr}(s)$ occurs in $regT_i$, then i belongs to s.
If $i \in \mathcal{D}$ and $\texttt{retAddr}(s)$ occurs in $opdT_i$, then $i = s$.

type frames assigned only to valid code indices (not necessarily to all of them)

initial type frame, assigned to 0:
declared meth arg types more
specific than the types in $regT_0$
(this in reg_0 of meth class type
c & partly initlized by constr)
opd is empty

successor type frame more specific than type frame assigned to succ index

Assume: a) compiled finally code is connected
b) subroutine starts with a Store(addr,x)

used for return by Ret(x)

retAddrs occur in regs only within subroutines, on stack only at its start

Type frames assigned to valid indices: conditions at 0 and at successors

Definition 16.3.8 (Bytecode type assignment). A bytecode type assignment with domain $\mathcal{D}$ for a method μ is a family $(regT_i, opdT_i)_{i\in\mathcal{D}}$ type frames satisfying the following conditions:

T1. $\mathcal{D}$ is a set of valid code indices of the method μ.

T2. Code index 0 belongs to $\mathcal{D}$.

T3. Let $[\tau_1, \dots, \tau_n] = argTypes(\mu)$ and $c = classNm(\mu)$. If μ is a
- a) class initialization method: $regT_0 = \emptyset$.
- b) class method: $\{0 \mapsto \tau_1, \dots, n-1 \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.
- c) instance method: $\{0 \mapsto c, 1 \mapsto \tau_1, \dots, n \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.
- d) constructor: $\{0 \mapsto InInit, 1 \mapsto \tau_1, \dots, n \mapsto \tau_n\} \sqsubseteq_{\text{reg}} regT_0$.

T4. The list $opdT_0$ is empty.

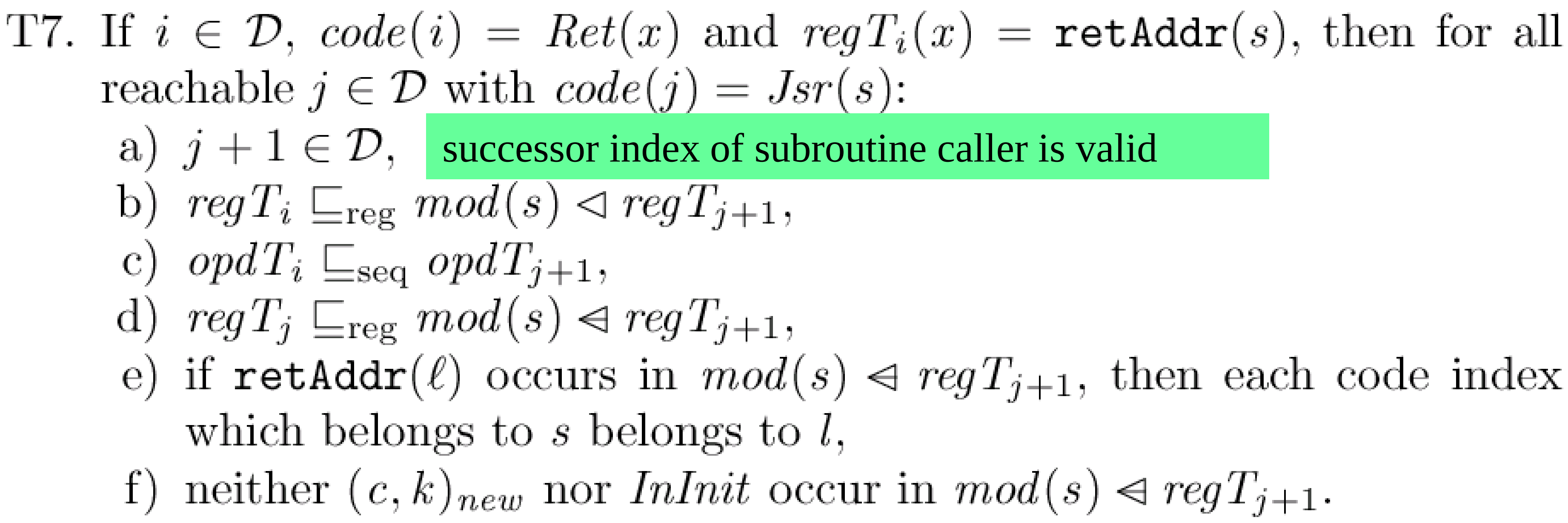
T5. If $i \in \mathcal{D}$, then $check(\mu, i, regT_i, opdT_i)$ is true.

T6. If $i \in \mathcal{D}$ and $(j, regS, opdS) \in succ(\mu, i, regT_i, opdT_i)$, then $j \in \mathcal{D}$, $regS \sqsubseteq_{\text{reg}} regT_j$ and $opdS \sqsubseteq_{\text{seq}} opdT_j$.

Type frames have to satisfy the check conditions

Subroutine type frame conditions upon return to successor of reachable subroutine caller:

type of local variables to be used at successor j+1 is less specific there than at return point i - if modified by the subroutine - , at caller j otherwise; type of **opd** at return point is more specific than at continuation point j+1



T7. If $i \in \mathcal{D}$, $code(i) = Ret(x)$ and $regT_i(x) = \texttt{retAddr}(s)$, then for all reachable $j \in \mathcal{D}$ with $code(j) = Jsr(s)$:

a) $j + 1 \in \mathcal{D}$,

successor index of subroutine caller is valid

b) $regT_i \sqsubseteq_{\text{reg}} mod(s) \lhd regT_{j+1}$,

c) $opdT_i \sqsubseteq_{\text{seq}} opdT_{j+1}$,

d) $regT_j \sqsubseteq_{\text{reg}} mod(s) \ntriangleleft regT_{j+1}$,

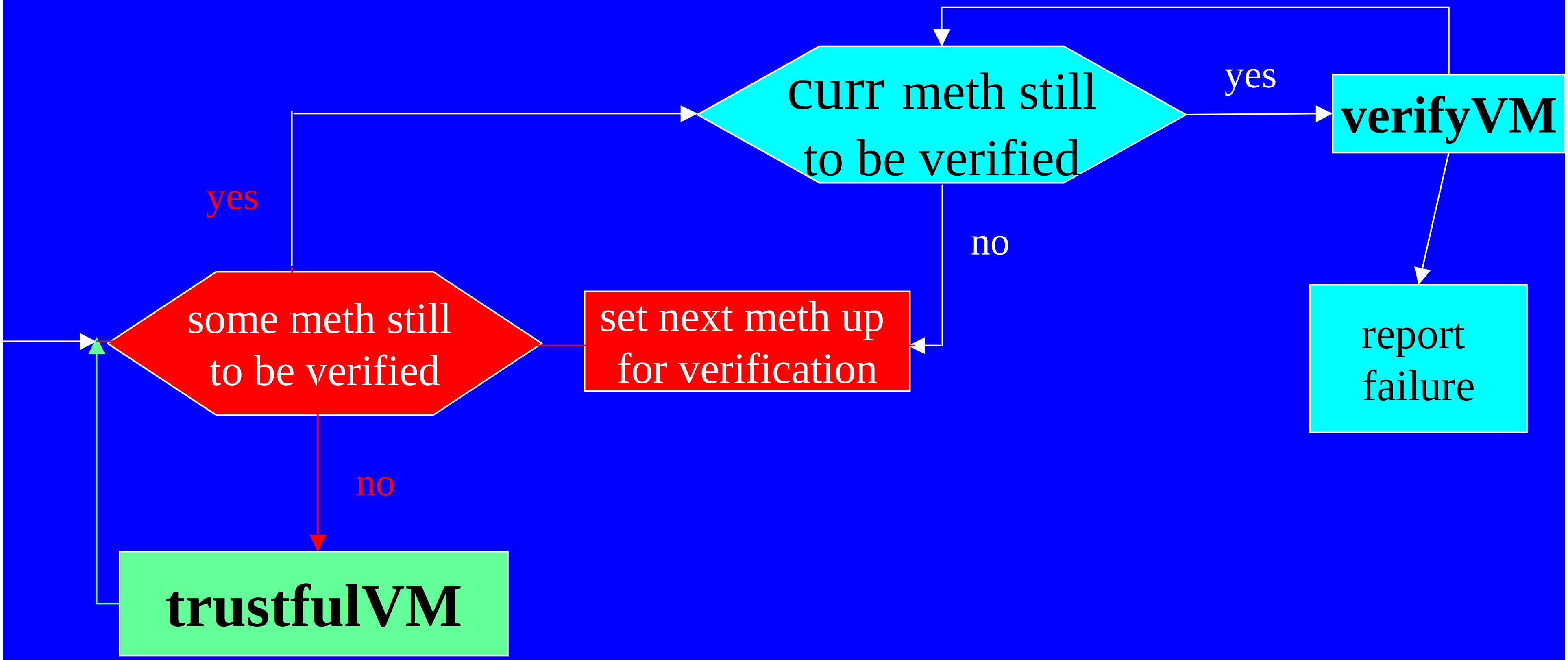
e) if $\texttt{retAddr}(\ell)$ occurs in $mod(s) \ntriangleleft regT_{j+1}$, then each code index which belongs to s belongs to l,

f) neither $(c,k)_{new}$ nor $InInit$ occur in $mod(s) \ntriangleleft regT_{j+1}$.

e) Proper nesting of subroutines: a retAddr occuring at succ of caller of a subroutine, which did not modify it, is addr of an enclosing subroutine

f) no not fully initialized object can be used at succ of caller of a subroutine without having been modified by the subroutine (guarantees that there is at most one type $(c,k)_{new}$ & prevents double initialization)

Stepwise refinement of diligentVM$_{I,C,O,E}$



switchVM$_C$ in trustfulVM is refined to also link classes before their initialization, where the linking submachine triggers verifyVM

verifyVM built out of langg layered check, succ, propagate

The state of the verifier

$regV_i$, $opdV_i$ to store register and opd stack types computed for instr i

Initially $opdV_o = []$, $regV_0$ = types of meth args and target ref, otherwise undefined

visited(i) indicating that to instr i a type frame has been associated

changed(i) for instrs i whose type frame has still to be checked before being propagated to successors

Initially $changed_o = visited_0$ = true, otherwise undef

verifyMeths: Class/MSig* $meth_v$ = top(verifyMeths) verifyClass

Def: some method still to be verified iff verifyMeths $\neq$ [ ]

curr method still to be verified iff dom (changed) $\neq \emptyset$

report failure = (halt : = FailureReport)

For correct propagation of type frames upon return from subroutines, two fcts **enterJsr** and **leaveJsr** are needed to record visited code indices where a subroutine has been entered or exited

Macros for initializing VerifyVM

set next meth up for verification

```
let verifyMeths’ = drop(verifyMeths, 1)
    verifyMeths := verifyMeths’
    if length(verifyMeths’) > 0 then
    initVerify(top(verifyMeths’))
    else
    classState(verifyClass) := Linked
```

initVerify(meth)

```
visited(0) := True
changed(0) := True
$regV_0$ := formals(meth)
$opdV_0$ := [ ]
forall i $\in$ dom(visited), i $\neq$0
    visited(i ) := undef
    changed(i ) := undef
    $regV_i$ := undef
    $opdV_i$ := undef
```

Type correctness of meth invocation is guaranteed by formals (meth), which initially assigns to the type registers the arg types of the meth and for inst meths/constructors also the type of the target reference (i.e. the class of the meth or InInit)

Linking classes before their initialization triggers their verification

switchVM$_C$ is extended by the rule

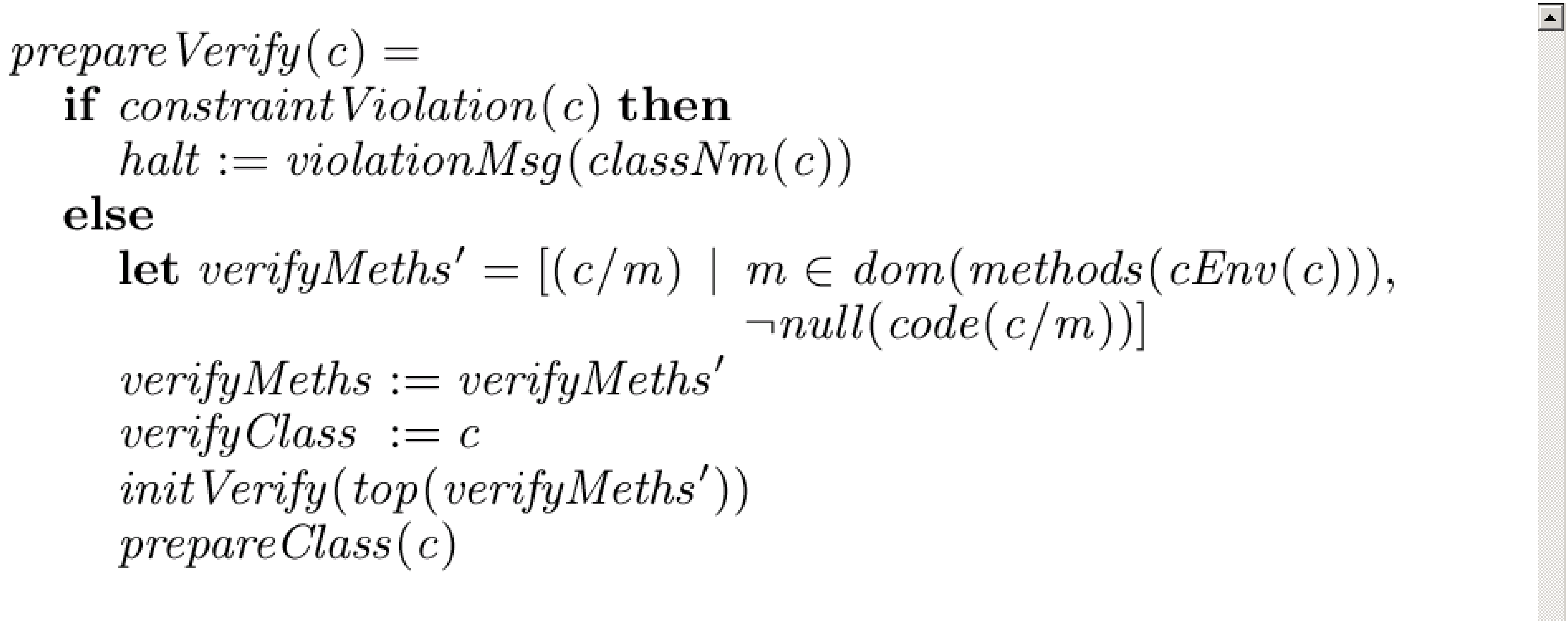
case switch of

InitClass(c) → if classState(c) = Referenced then linkClass(c)

```
$linkClass(c) =$
  let $classes = \{super(c)\} \cup implements(c)$
  if $c = \texttt{Object} \lor \forall c' \in classes : classState(c') \geq Linked$ then
    $prepareVerify(c)$
  elseif $\neg cyclicInheritance(c)$ then
    choose $c' \in classes, classState(c') = Referenced$
      $linkClass(c')$
  else
    $halt := \texttt{"Cyclic Inheritance: "} \cdot classNm(c)$
```

This recursive submachine terminates since the class inheritance hierarchy is finite

The preparatory test checks the class format of the class file and the static constraints for the method bodies



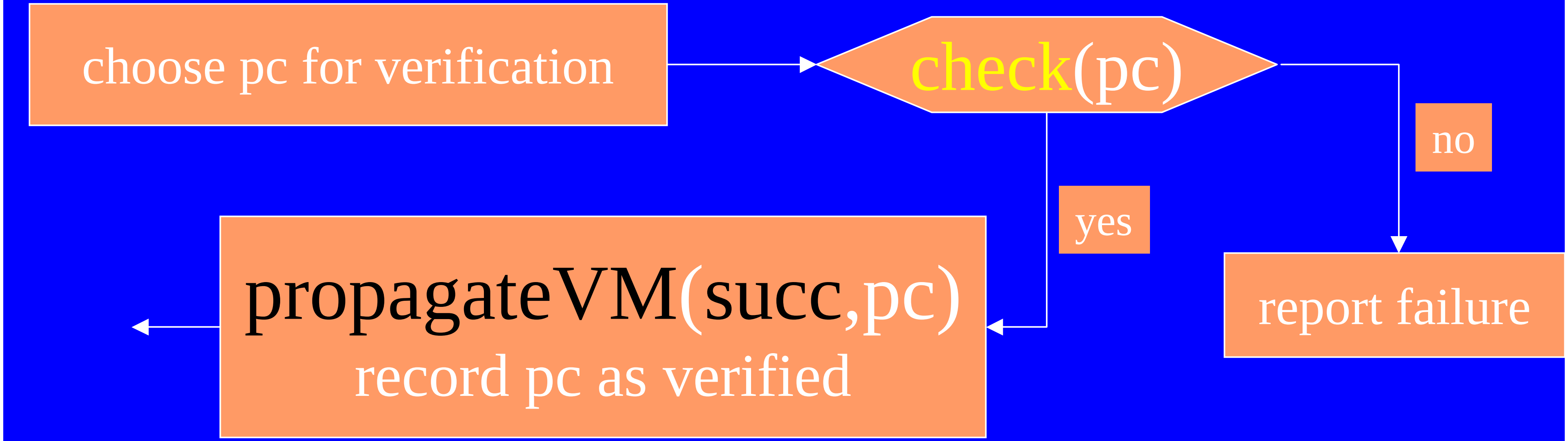
```
$prepareVerify(c) =$
  if $constraintViolation(c)$ then
    $halt := violationMsg(classNm(c))$
  else
    let $verifyMeths' = [(c/m) \mid m \in dom(methods(cEnv(c))), \neg null(code(c/m))]$
    $verifyMeths := verifyMeths'$
    $verifyClass := c$
    $initVerify(top(verifyMeths'))$
    $prepareClass(c)$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

constraintViolation checks class file format and other static conditions imposed on the method bodies

Class preparation macro to create and initialize static fields

```
prepareClass(c) =
        forall f ∈ staticFields(c)
                globals(c/f ) := defaultVal (type(c/f ))
```

Stepwise refinement of $\text{verifyVM}_{I,C,O,E}$



choose pc for verification

check(pc)

no

yes

report failure

propagateVM(succ,pc)

record pc as verified

Defn. choose pc for verification = choose pc ∈ dom(changed)

record pc as verified = (changed(pc) := undef)

propagateVM the checked type frame from pc to all possible successor frames, simulating execVM on types frames

Stepwise refinement: $\text{propagate}_I \subseteq \text{propagate}_E$

(no propagation for native meths) $\text{succ}_I \subseteq \text{succ}_C \subseteq \text{succ}_O \subseteq \text{succ}_E$

Computing successor frames by simulating execVM$_I$ on types (reg/opd weakly typed)

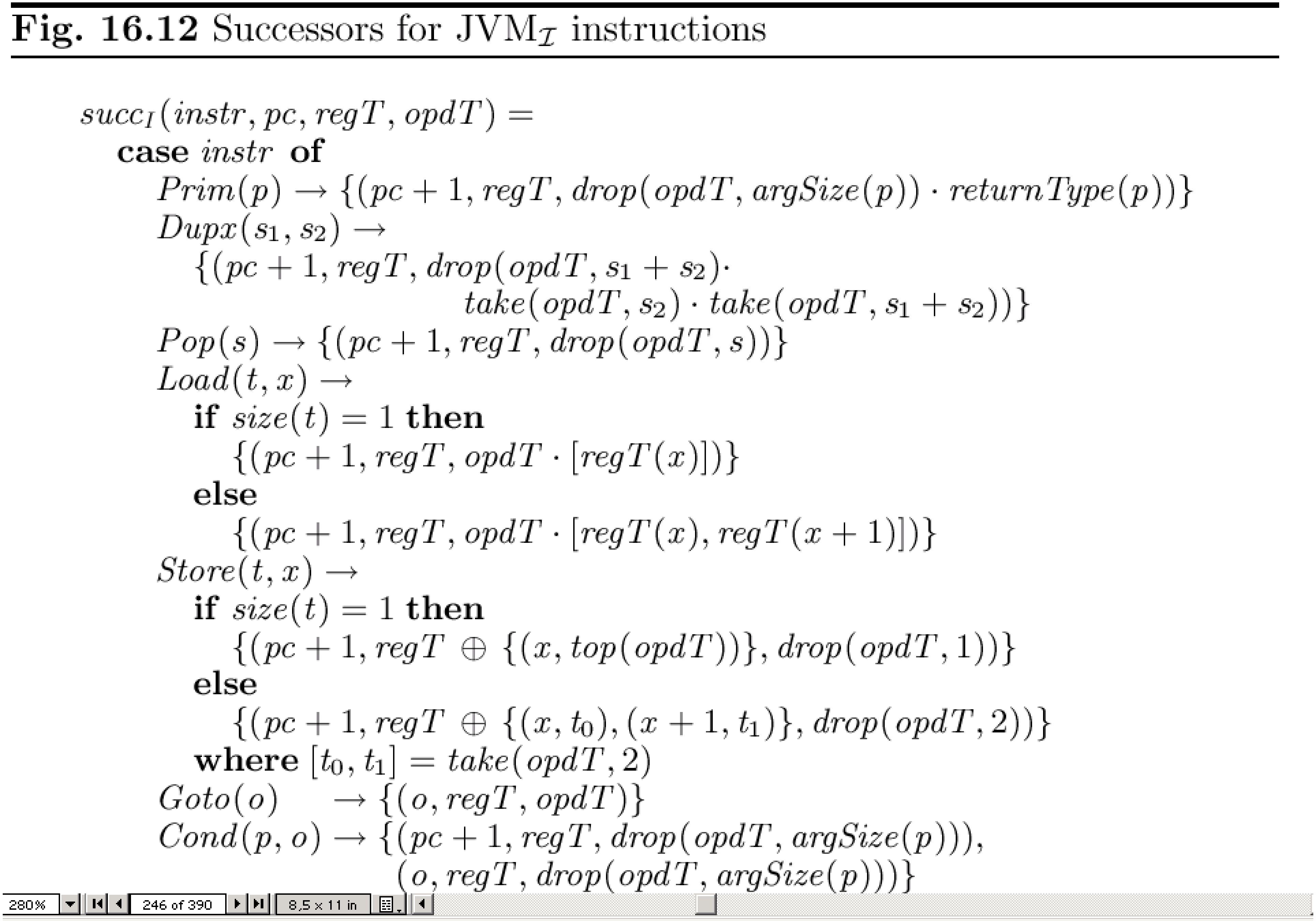
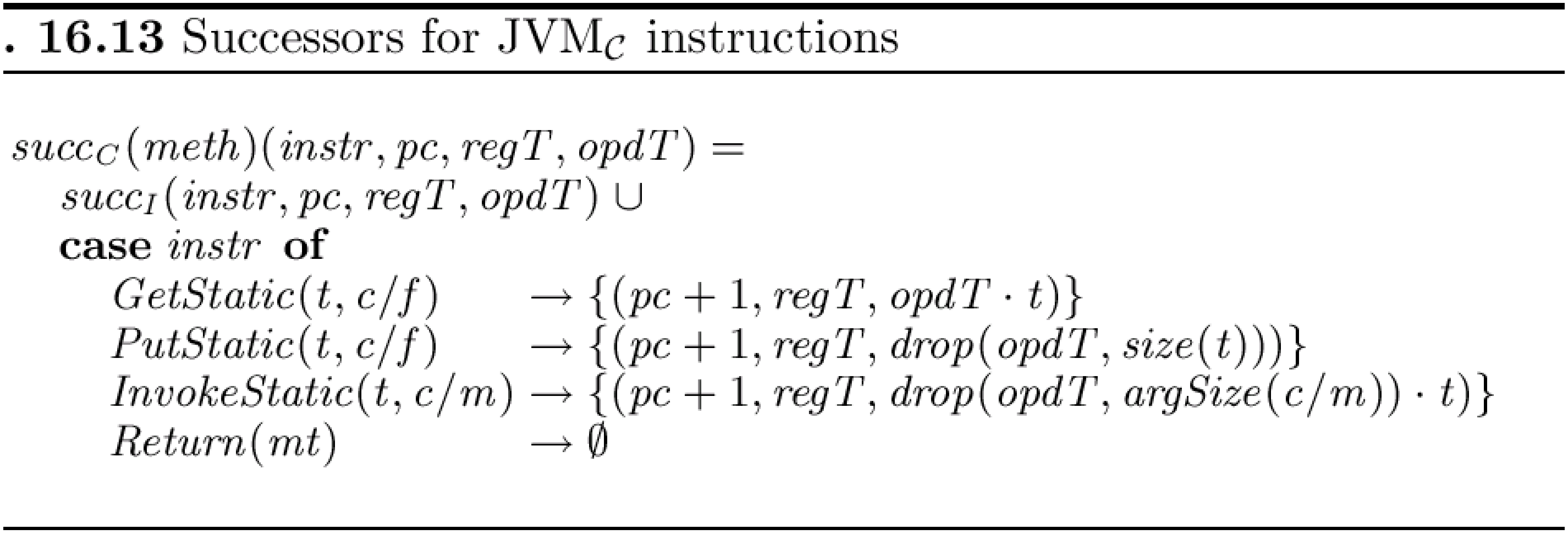


Fig. 16.12 Successors for JVM$_\mathcal{I}$ instructions

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

Extending successor type frames by simuln of execVM$_C$ instrs



. 16.13 Successors for JVM$_{\mathcal{C}}$ instructions

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

NB: Class fields are stronlgy typed, holding always only one single type (differently from reg and opd). Unlike the DefensiveVM, VerifyVM therefore uses the declared type of the global field (stored as instr param).

Similarly for class meth invocs, the declared return type is propagated.

Return instrs generate no successor (in the method they leave)

Link-time checkable requiremts on objects & their initialization

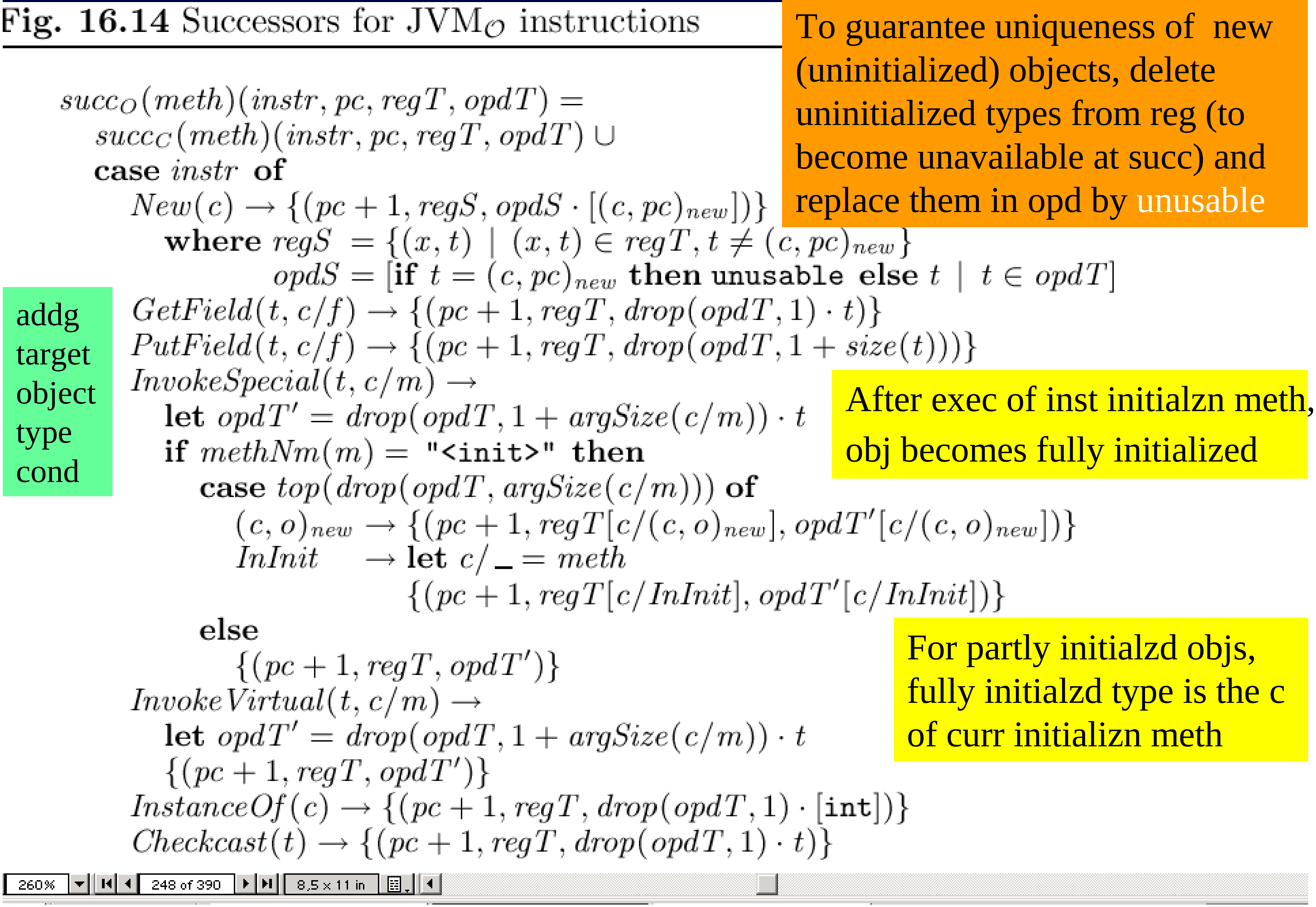


Fig. 16.14 Successors for JVM$_{\mathcal{O}}$ instructions

```
$succ_O(meth)(instr, pc, regT, opdT) =$
  $succ_C(meth)(instr, pc, regT, opdT) \cup$
  $\mathbf{case}\ instr\ \mathbf{of}$
    $New(c) \rightarrow \{(pc+1, regS, opdS \cdot [(c, pc)_{new}])\}$
      $\mathbf{where}\ regS = \{(x, t) \mid (x, t) \in regT, t \neq (c, pc)_{new}\}$
                $opdS = [\mathbf{if}\ t = (c, pc)_{new}\ \mathbf{then\ unusable\ else}\ t \mid t \in opdT]$
    $GetField(t, c/f) \rightarrow \{(pc+1, regT, drop(opdT, 1) \cdot t)\}$
    $PutField(t, c/f) \rightarrow \{(pc+1, regT, drop(opdT, 1 + size(t)))\}$
    $InvokeSpecial(t, c/m) \rightarrow$
      $\mathbf{let}\ opdT' = drop(opdT, 1 + argSize(c/m)) \cdot t$
      $\mathbf{if}\ methNm(m) = \texttt{"<init>"}\ \mathbf{then}$
        $\mathbf{case}\ top(drop(opdT, argSize(c/m)))\ \mathbf{of}$
          $(c, o)_{new} \rightarrow \{(pc+1, regT[c/(c, o)_{new}], opdT'[c/(c, o)_{new}])\}$
          $InInit \rightarrow \mathbf{let}\ c/\_ = meth$
                      $\{(pc+1, regT[c/InInit], opdT'[c/InInit])\}$
        $\mathbf{else}$
          $\{(pc+1, regT, opdT')\}$
    $InvokeVirtual(t, c/m) \rightarrow$
      $\mathbf{let}\ opdT' = drop(opdT, 1 + argSize(c/m)) \cdot t$
      $\{(pc+1, regT, opdT')\}$
    $InstanceOf(c) \rightarrow \{(pc+1, regT, drop(opdT, 1) \cdot [\mathtt{int}])\}$
    $Checkcast(t) \rightarrow \{(pc+1, regT, drop(opdT, 1) \cdot t)\}$
```

To guarantee uniqueness of new (uninitialized) objects, delete uninitialized types from reg (to become unavailable at succ) and replace them in opd by unusable

addg target object type cond

After exec of inst initialzn meth, obj becomes fully initialized

For partly initialzd objs, fully initialzd type is the c of curr initializn meth

Determining handler frames for successors of JVM$_E$ instrs

Fig. 16.15 Successors for JVM$_{\mathcal{E}}$ instructions

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

Every handler in exception table yields a possible successor

Ret taken into account by defn of type assignment, with types of local vars propagated both from the subroutine return index and from successor index of subroutine call

We assume Jsr(_), Goto(_), Return(_), Load(_,_), which are used for the compilation of abruption (jump and return) stms, not to throw exceptions so that allhandlers(instr,m, pc, regT) = ∅ , otherwise we include into successors all handlers which protect the code index (for instr = code(pc)):

allhandlers(instr, m, pc, regT) =

$$\{(h, regT, [t\,]) \mid (f, u, h, t) \in excs(m) \;\&\; f \le pc < u\}$$

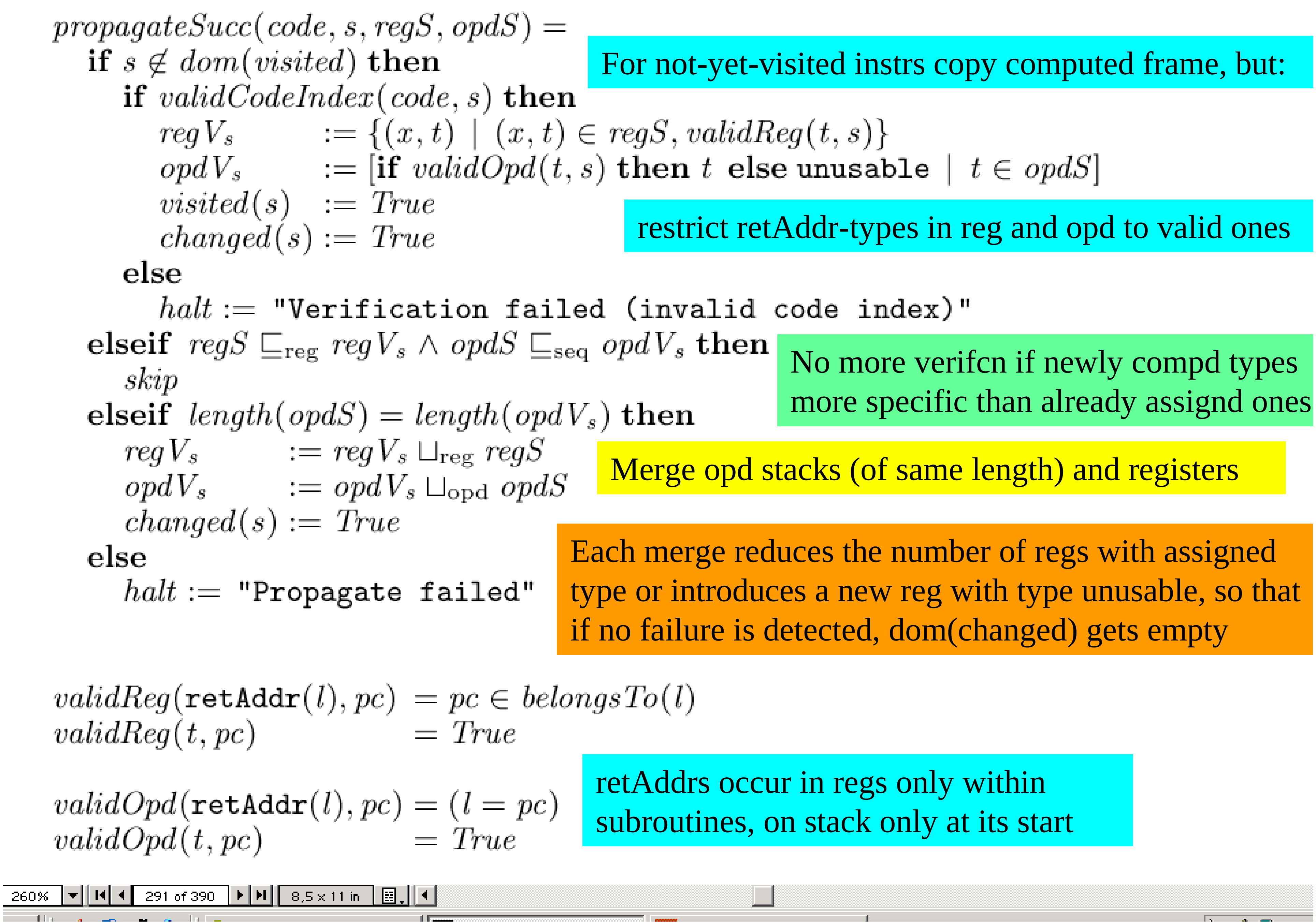
Type reg/opd propagation to successors

```
**propagateVM$_I$** (code, succ, pc) =

forall (s, regS, opdS) $\in$
                    succ(code(pc), pc, regV$_{pc}$, opdV$_{pc}$)
propagateSucc(code, s, regS, opdS)
```

Adding constraints for excs & embedded subroutines

```
**propagateVM$_E$** (code, succ, pc) =

propagateVM$_I$ (code, succ, pc)
propagateJsrRet(code, succ, pc)
```

Propagating type frames (regS,opdS) computed by succ to successor code indices s



```
$propagateSucc(code, s, regS, opdS) =$
  if $s \notin dom(visited)$ then
    if $validCodeIndex(code, s)$ then
      $regV_s := \{(x,t) \mid (x,t) \in regS, validReg(t,s)\}$
      $opdV_s := [\texttt{if } validOpd(t,s) \texttt{ then } t \texttt{ else unusable} \mid t \in opdS]$
      $visited(s) := True$
      $changed(s) := True$
    else
      $halt :=$ "Verification failed (invalid code index)"
  elseif $regS \sqsubseteq_{reg} regV_s \land opdS \sqsubseteq_{seq} opdV_s$ then
    $skip$
  elseif $length(opdS) = length(opdV_s)$ then
    $regV_s := regV_s \sqcup_{reg} regS$
    $opdV_s := opdV_s \sqcup_{opd} opdS$
    $changed(s) := True$
  else
    $halt :=$ "Propagate failed"
```

For not-yet-visited instrs copy computed frame, but:

restrict retAddr-types in reg and opd to valid ones

No more verifcn if newly compd types more specific than already assignd ones

Merge opd stacks (of same length) and registers

Each merge reduces the number of regs with assigned type or introduces a new reg with type unusable, so that if no failure is detected, dom(changed) gets empty

$$\begin{aligned} validReg(\texttt{retAddr}(l), pc) &= pc \in belongsTo(l) \\ validReg(t, pc) &= True \end{aligned}$$

$$\begin{aligned} validOpd(\texttt{retAddr}(l), pc) &= (l = pc) \\ validOpd(t, pc) &= True \end{aligned}$$

retAddrs occur in regs only within subroutines, on stack only at its start

Propagating type frames upon return to direct successors j+1 of any (reachable) j from where subroutine s can be entered

propagateJsrRet(code, succ, pc) =

```
case $code(pc)$ of
    $Jsr(s) \rightarrow enterJsr(s) := \{pc\} \cup enterJsr(s)$
        forall $(i, x) \in leaveJsr(s), i \notin dom(changed)$
            if $regV_i(x) = \texttt{retAddr}(s)$ then
                $propagateJsr(code, pc, s, i)$
    $Ret(x) \rightarrow$ let $\texttt{retAddr}(s) = regV_{pc}(x)$
        $leaveJsr(s) := \{(pc, x)\} \cup leaveJsr(s)$
        forall $j \in enterJsr(s), j \notin dom(changed)$
            $propagateJsr(code, j, s, pc)$
```

update enterJsr(s)

propagate to pc+1 types from correspndg subroutine returns i

update leaveJsr(s)

propagate types to j+1 for each corresponding subroutine entry j

enterJsr(s) = the set of visited indices of instrs Jsr(s)

leaveJsr(s) = set comprising all visited indices of instrs Ret(x) which assign type retAddr(s) to reg x

both functions are initialized in initVerify by ∅

Propagating types to direct successor j+1 of a subroutine call Jsr(s)

$$\begin{aligned}
&propagateJsr(code, j, s, i) = \\
&\quad propagateSucc(code, j+1, regJ \oplus mod(s) \lhd regV_i, opdV_i)\ \textbf{where} \\
&\qquad regJ = \{(x,t) \mid (x,t) \in mod(s) \mathbin{⩤} regV_j, \\
&\qquad\qquad\qquad validJump(t,s) \wedge t \neq (\_,\_)_{new} \wedge t \neq InInit\}
\end{aligned}$$

$$\begin{aligned}
validJump(\texttt{retAddr}(l), s) &= belongsTo(s) \subseteq belongsTo(l) \\
validJump(t, s) &= True
\end{aligned}$$

a) Restrict registers from the caller frame at j, which have not been modified by the subroutine s but will be used at j+1:

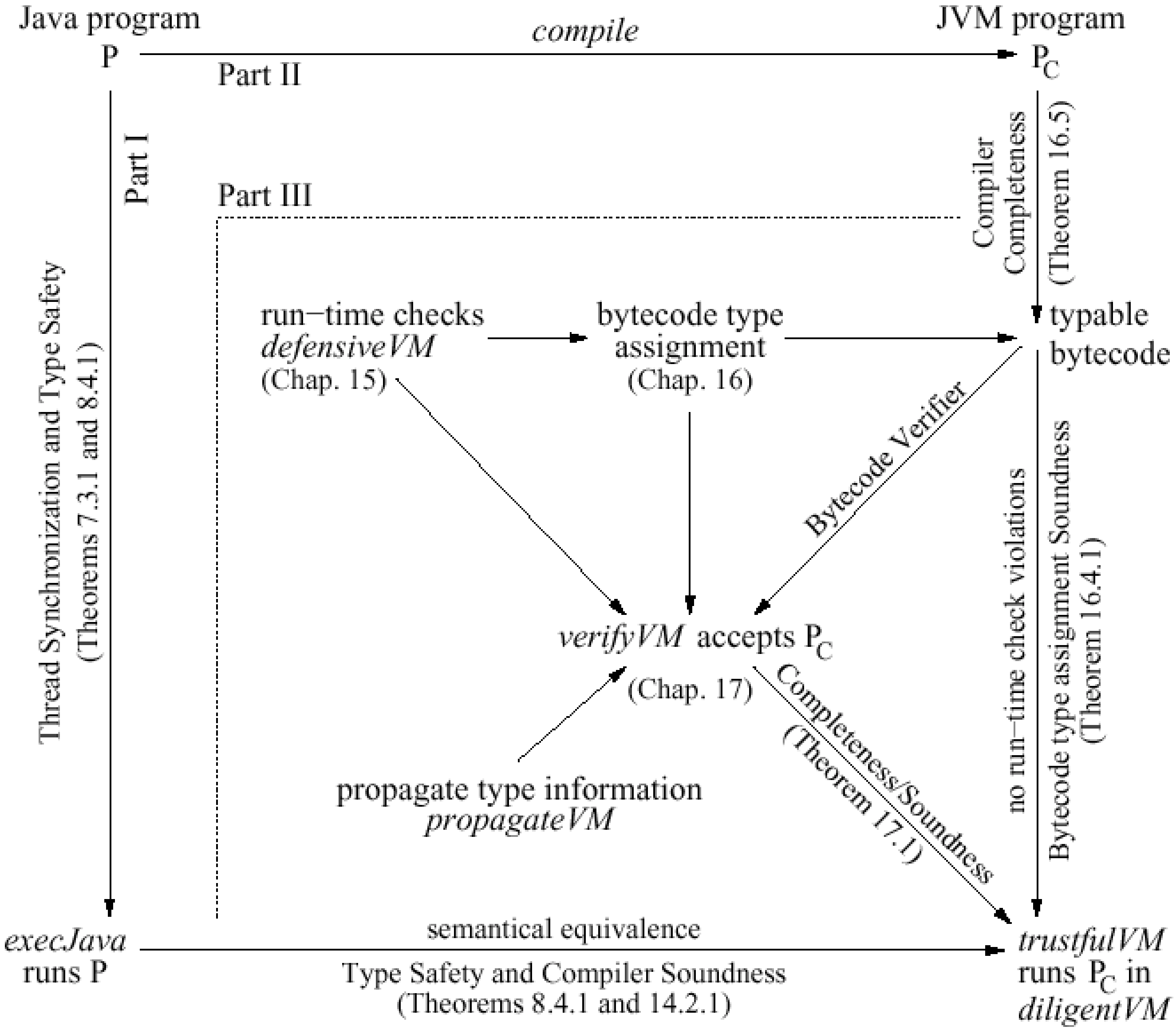
- for proper nesting of subroutines: to validJump types – i.e. of addresses of enclosing subroutines,
- for uniqueness of new (uninitialized) objects: to those of completely initialized objects.

b) Restrict registers from the return frame, which will be used at j+1, to those which have been modified by the subroutine s.

Proving Bytecode Verifier Complete and Correct

- Bytecode Verifier Soundness Theorem: For any program P, the Bytecode Verifier either rejects P or during the verification satisfies the type assignment conditions for P.
- Bytecode Verifier Completeness Theorem: If P has a type assignment, then the Bytecode Verifier does not reject P and computes a most specific type assignment.

Synopsis of machine decomposition and proof structuring



Java program
P

compile

Part II

JVM program
P_C

Part I

Part III

Compiler Completeness (Theorem 16.5)

Thread Synchronization and Type Safety (Theorems 7.3.1 and 8.4.1)

run−time checks
defensiveVM
(Chap. 15)

bytecode type
assignment
(Chap. 16)

typable
bytecode

Bytecode Verifier

verifyVM accepts P_C
(Chap. 17)

propagate type information
propagateVM

Completeness/Soundness
(Theorem 17.1)

no run−time check violations

Bytecode type assignment Soundness
(Theorem 16.4.1)

execJava
runs P

semantical equivalence

Type Safety and Compiler Soundness
(Theorems 8.4.1 and 14.2.1)

trustfulVM
runs P_C in
diligentVM

Dependency Graph of the book chapters

Dynamic Loading (finding binary form) & Linking (preparation and verification) integrated into run-time

by extension execVM_D for loader meths & $\mathrm{switchVM}_D$ to reference loaded classes and superclasses before linking

Classes extended by loader, which provides name space (for all types):

Class = (Ld,Name)

ldEnv:Class → Ref yields the class object loaded by given loader under given name

cOf: Ref → Class yields the class name with its defining (maybe ≠ initiating) loader

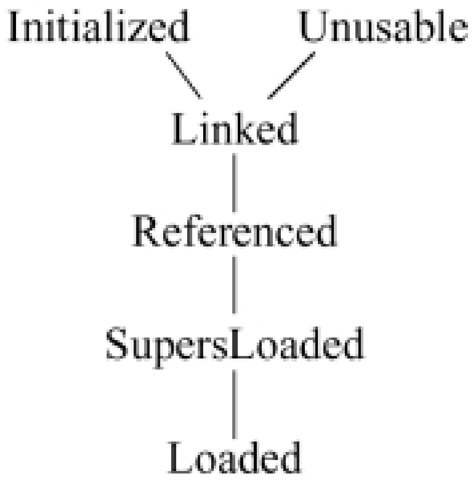
liftClass(c) = cOf(ldEnv(c)) yields the defining loader

cEnv:Class → ClassFile dynamic fct

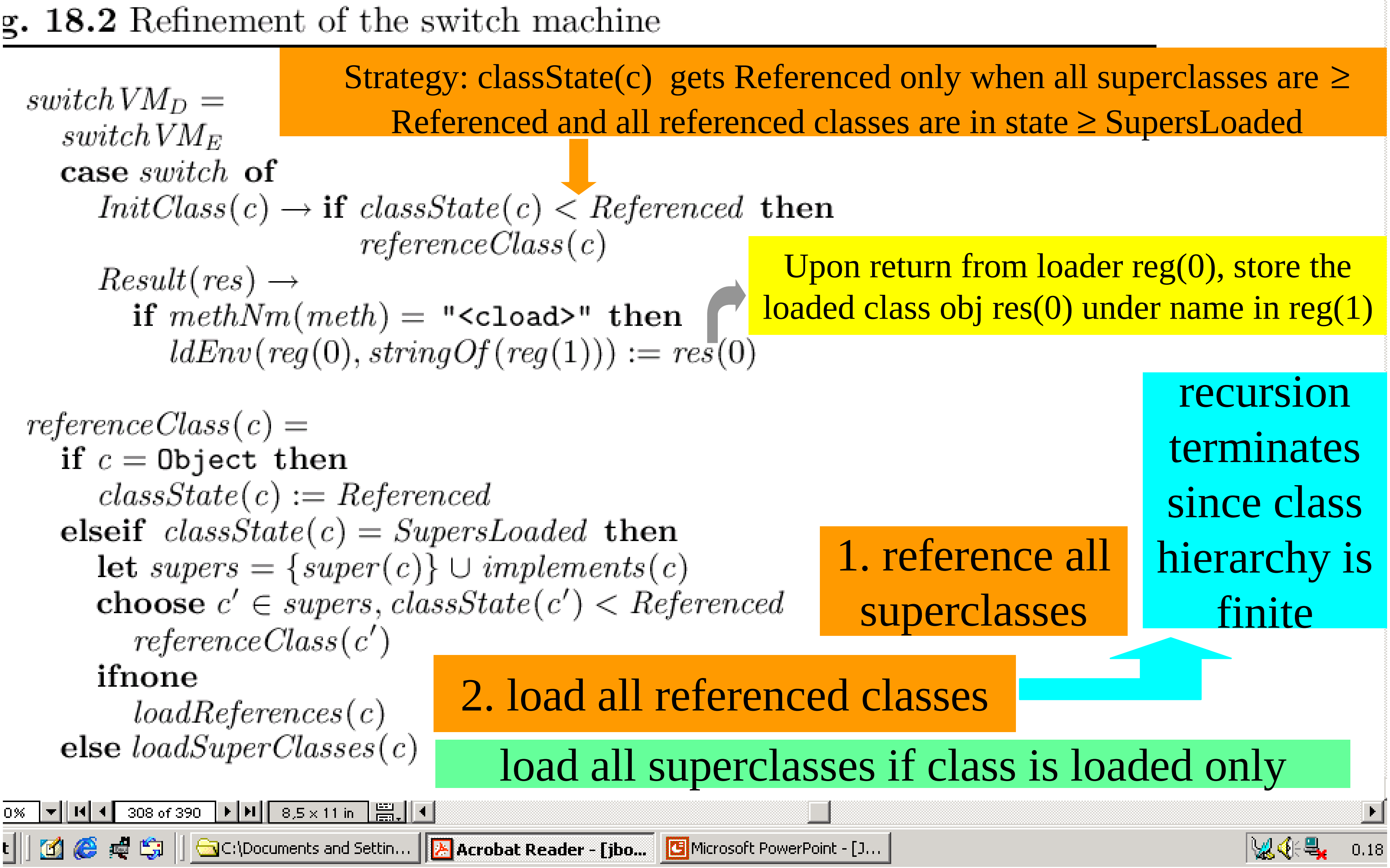
classState(c)=Loaded means c is loaded

classState(c)=SupersLoaded means all superclasses loaded with classState ≥ SupersLoaded

classState(c)=Referenced means all superclasses have classState ≥ Referenced and all referenced classes have classState ≥ SupersLoaded



Task: guarantee the complete availability of all types which may occur during execution of a loaded class



g. 18.2 Refinement of the switch machine

```
$switchVM_D =$
  $switchVM_E$
  case $switch$ of
    $InitClass(c) \rightarrow$ if $classState(c) < Referenced$ then
                        $referenceClass(c)$
    $Result(res) \rightarrow$
      if $methNm(meth) =$ "<cload>" then
        $ldEnv(reg(0), stringOf(reg(1))) := res(0)$

$referenceClass(c) =$
  if $c =$ Object then
    $classState(c) := Referenced$
  elseif $classState(c) = SupersLoaded$ then
    let $supers = \{super(c)\} \cup implements(c)$
    choose $c' \in supers, classState(c') < Referenced$
      $referenceClass(c')$
    ifnone
      $loadReferences(c)$
  else $loadSuperClasses(c)$
```

Strategy: classState(c) gets Referenced only when all superclasses are ≥ Referenced and all referenced classes are in state ≥ SupersLoaded

Upon return from loader reg(0), store the loaded class obj res(0) under name in reg(1)

recursion terminates since class hierarchy is finite

1. reference all superclasses

2. load all referenced classes

load all superclasses if class is loaded only

Implicit callLoad (ld,cn) = (switch := Call (<cload>, [ld,cn]))

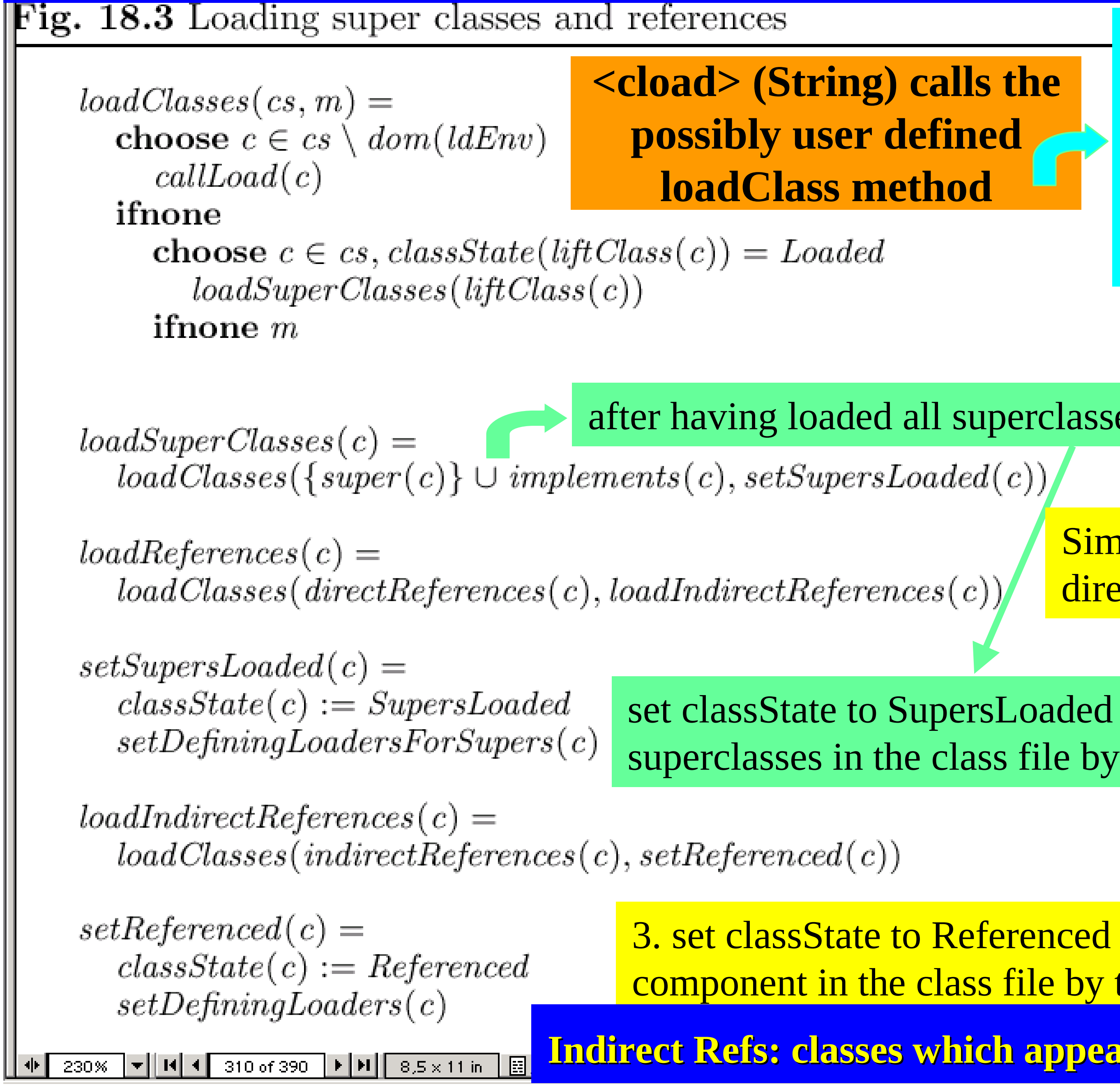


Fig. 18.3 Loading super classes and references

```
$loadClasses(cs, m) =$
    choose $c \in cs \setminus dom(ldEnv)$
        $callLoad(c)$
    ifnone
        choose $c \in cs, classState(liftClass(c)) = Loaded$
            $loadSuperClasses(liftClass(c))$
        ifnone $m$

$loadSuperClasses(c) =$
    $loadClasses(\{super(c)\} \cup implements(c), setSupersLoaded(c))$

$loadReferences(c) =$
    $loadClasses(directReferences(c), loadIndirectReferences(c))$

$setSupersLoaded(c) =$
    $classState(c) := SupersLoaded$
    $setDefiningLoadersForSupers(c)$

$loadIndirectReferences(c) =$
    $loadClasses(indirectReferences(c), setReferenced(c))$

$setReferenced(c) =$
    $classState(c) := Referenced$
    $setDefiningLoaders(c)$
```

<cload> (String) calls the possibly user defined loadClass method

Load(addr, 0) loader
Load(addr, 1) class name
InvokeVirtual
 (Class,loadClass(String))
Return(addr)

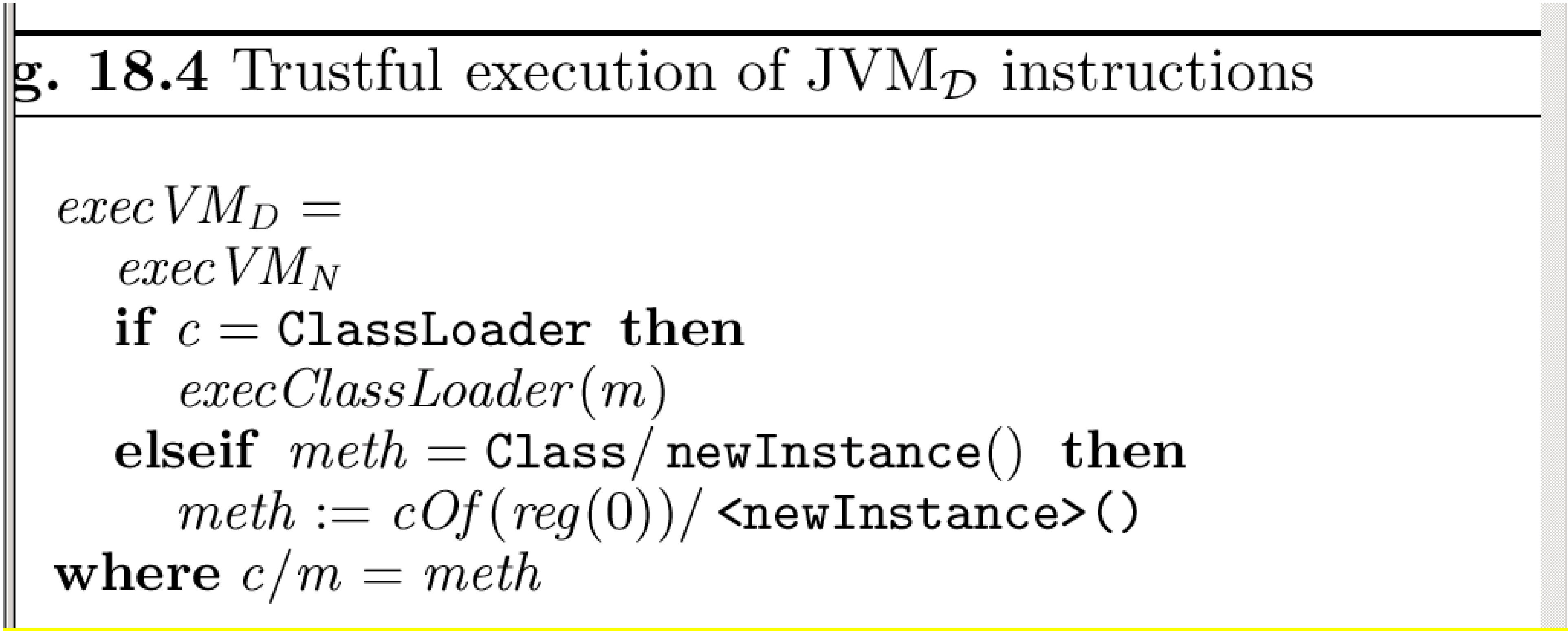
after having loaded all superclasses

Similarly for references: 1. load direct refs, 2. load indirect refs,

set classState to SupersLoaded and replace loader of superclasses in the class file by the defining loader

3. set classState to Referenced and replace loader component in the class file by the defining loader

Indirect Refs: classes which appear in context of other refs



g. 18.4 Trustful execution of $JVM_{\mathcal{D}}$ instructions

```
$execVM_D =$
    $execVM_N$
    if $c$ = ClassLoader then
        $execClassLoader(m)$
    elseif $meth$ = Class/newInstance() then
        $meth := cOf(reg(0))/$<newInstance>()
where $c/m = meth$
```

Extension of $execVM_N$ by native methods for

a) class loading/resolving

b) newInstance to create a new instance for a class object

Refine correspondingly $check_N$ for $defensiveVM_D$ and $diligentVM_D$ to recognize also native methods for dynamic loading:

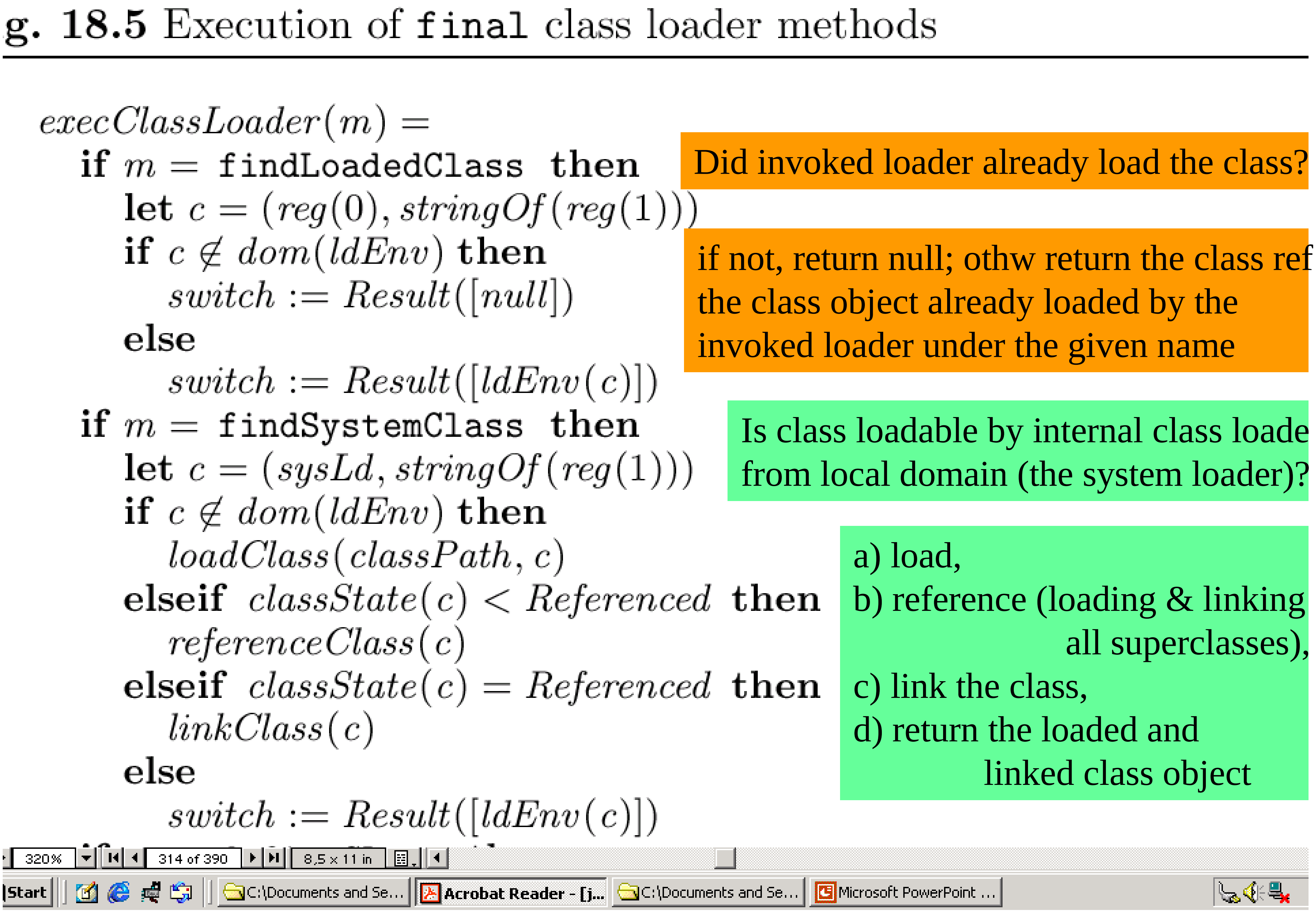
$check_D(c/m)$ =

c = ClassLoader & m∈{findLoadedClass, findSystemClass, resolveClass, defineClass}

or c / m = Class / newInstance()

or $check_N(c / m)$

Loading attempted by invoked loader, internal loader, non-locally



g. 18.5 Execution of `final` class loader methods

```
$execClassLoader(m) =$
  if $m =$ findLoadedClass then
    let $c = (reg(0), stringOf(reg(1)))$
    if $c \notin dom(ldEnv)$ then
      $switch := Result([null])$
    else
      $switch := Result([ldEnv(c)])$
  if $m =$ findSystemClass then
    let $c = (sysLd, stringOf(reg(1)))$
    if $c \notin dom(ldEnv)$ then
      $loadClass(classPath, c)$
    elseif $classState(c) < Referenced$ then
      $referenceClass(c)$
    elseif $classState(c) = Referenced$ then
      $linkClass(c)$
    else
      $switch := Result([ldEnv(c)])$
```

Did invoked loader already load the class?

if not, return null; othw return the class ref
the class object already loaded by the
invoked loader under the given name

Is class loadable by internal class loade
from local domain (the system loader)?

a) load,
b) reference (loading & linking
all superclasses),
c) link the class,
d) return the loaded and
linked class object

Loading attempted by invoked loader, internal loader, non-locally

```
if $m$ = defineClass then
    let $c = (reg(0), stringOf(reg(1)))$
    if $c \notin dom(ldEnv)$ then
        let $content = arrayContent(heap(reg(2)), reg(3), reg(4))$
        $defineClass(content, c, True)$
    else
        $raise($"ClassFormatError"$)$
if $m$ = resolveClass then
    let $r = reg(1)$
    if $r = null$ then
        $raise($"NullPointerException"$)$
    else
        let $c = cOf(r)$
        if $classState(c) < Referenced$ then
            $referenceClass(c)$
        elseif $classState(c) = Referenced$ then
            $linkClass(c)$
        else
            $switch := Result([\,])$
```

If no local class was found

Check that class name not already in loader name space

read bytecode from origin of referenced class & create & return class object (without referencing or linking yet)

implicitly called before initializing a class

reference and link the class specified by the ref of the class object

Macros for Loading, Defining, and Linking classes

g. 18.6 Loading and linking machines

```
$loadClass(classPath, c) =$
    if $c \notin dom(load(classPath))$ then
        $raise($"ClassNotFoundException"$)$
    else
        $defineClass(load(classPath), c, False)$
```

Check whether the class exists in the local file system

```
$defineClass(content, c, returnClass) =$
    let $cf = analyze(content)$
    if $classNm(cf) \neq classNm(c)$ then
        $raise($"ClassFormatError"$)$
    else create $r$
        $classState(c) := Loaded$
        $heap(r)       := Object($Class$, \emptyset)$
        $cOf(r)        := c$
        $cEnv(c)       := cf$
        $ldEnv(c)      := r$
        if $returnClass$ then $switch := Result([r])$
```

check whether the class name coincides with the expected one

create a new class object and initialize its dynamic functions

Macros for Loading, Defining, and Linking classes

```
$linkClass(c) =$
    let $classes = \{super(c)\} \cup implements(c)$
    if $c = \texttt{Object} \lor \forall c' \in classes : classState(c') \geq Linked$ then
        $classState(c) := Linked$
        $prepareClass(c)$
    elseif $\neg cyclicInheritance(c)$ then
        choose $c' \in classes, classState(c') = Referenced$
            $linkClass(c')$
    else
        $halt :=$ "Cyclic Inheritance: " $\cdot\ classNm(c)$
```

The recursive submachine linkClass terminates because of the finiteness of the class hierarchy.

NB. Same machine linkClass as in switchVM$_C$ **except for using only the submachine prepareClass of prepareVerify:**

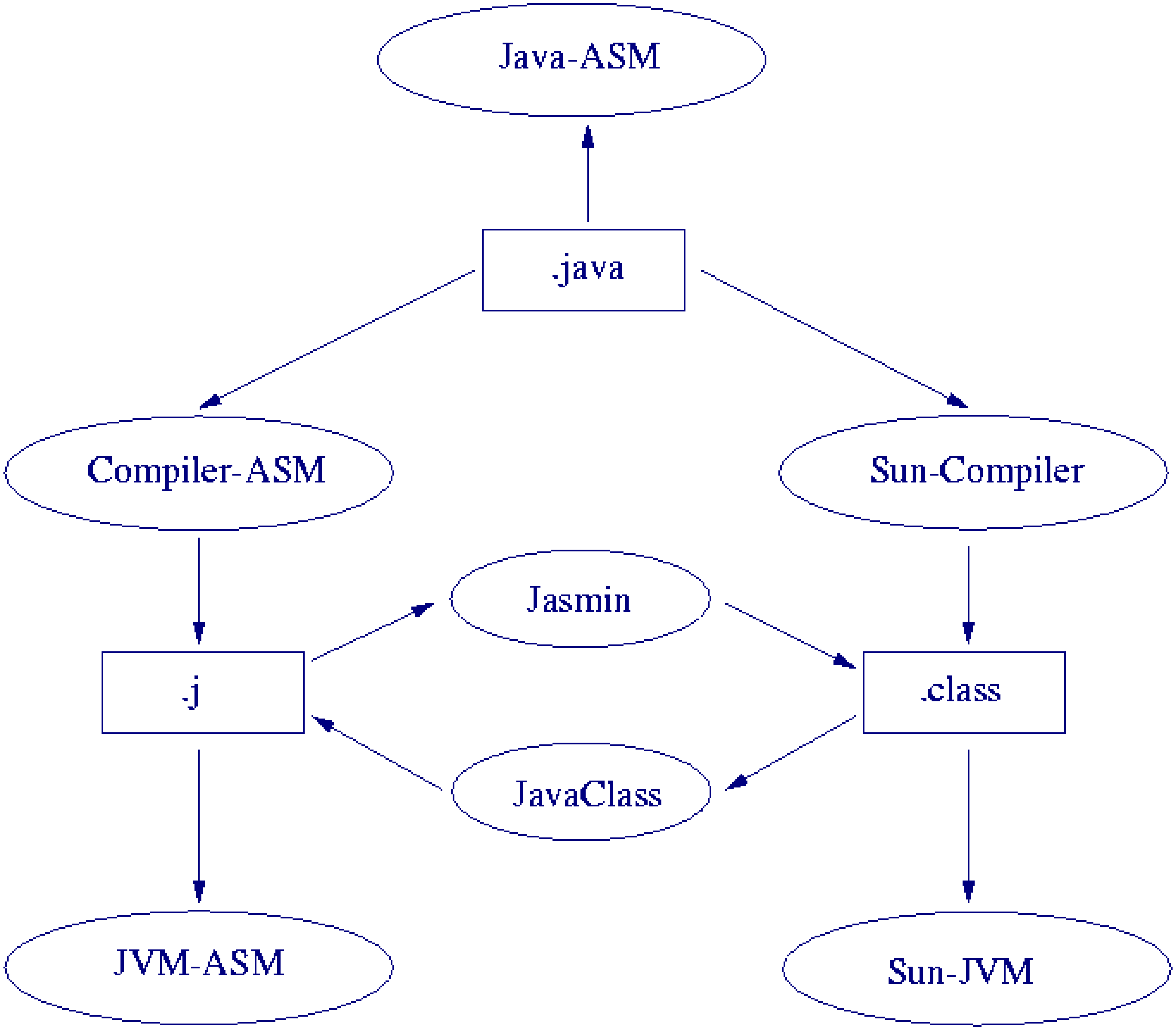
prepareClass(c) = forall f $\in$ staticFields(c)

globals(c/f) := defaultVal (type(c/f))

Validating Java, JVM, compile

- AsmGofer: ASM programming system, extending TkGofer to execute ASMs (with Haskell definable external fcts)
- Provides step-by-step execution, with GUIs to support debugging of Java/JVM programs.
- Allows for the executable ASM models of Java/JVM:
  - to execute the Java source code P (no counterpart in SUN env)
  - to compile Java pgms P to bytecode compile(P) (in textual representation, using JASMIN to convert to binary class format)
  - to execute the bytecode programs compile(P)

  E.g. our Bytecode Verifier rejects Saraswat's program
- Developed by Joachim Schmid, available at `www.tydo.de/AsmGofer`


Java-ASM
.java
Compiler-ASM
Sun-Compiler
Jasmin
.j
.class
JavaClass
JVM-ASM
Sun-JVM

Reference:

Java and the Java Virtual Machine. Definition, Verification, Validation

R. Stärk, J. Schmid, E. Börger

Springer-Verlag , 2001.

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