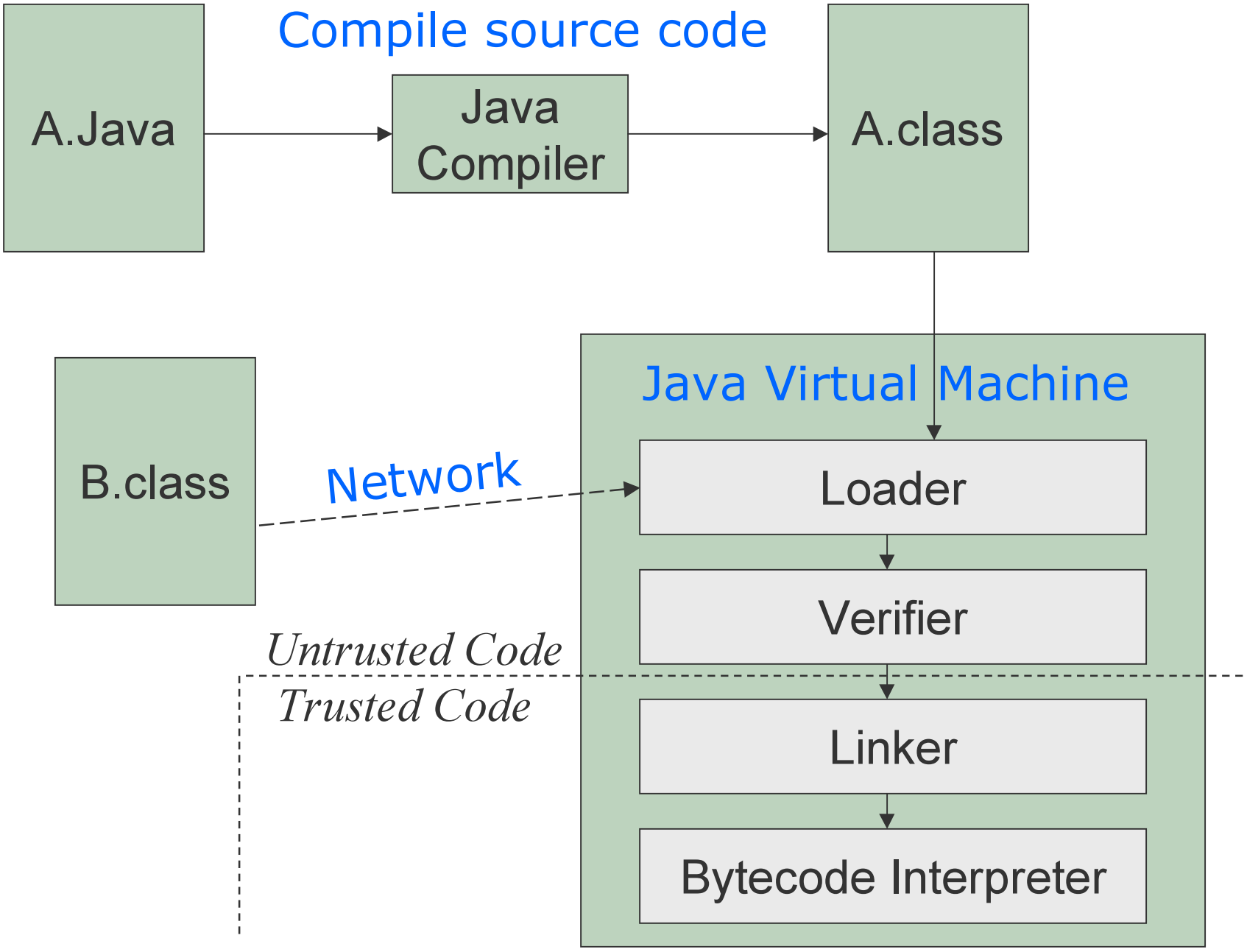
A Type System for Object Initialization in the Java Bytecode Language

Stephen Freund John Mitchell

Stanford University

Outline

- Java Virtual Machine
- JVML_i framework [Stata,Abadi 98]
- Object initialization
- JVML_i type rules
- Results
  - Soundness theorem
  - Previously unpublished bug in Sun's verifier


A.Java
Compile source code
Java Compiler
A.class
B.class
Network
Java Virtual Machine
Loader
Verifier
Untrusted Code
Trusted Code
Linker
Bytecode Interpreter

Java Bytecodes

- Java:

```
class A extends Object {
  int i;
  void f(int val) { i = val + 1; }
}
```

- JVML_i:

```
Method void f(int)
  0 aload_0
  1 iload_1
  2 iconst_1
  3 iadd
  4 putfield #4 <Field int i>
  5 return
```

The Verifier

- Guarantees that a program will not cause an unchecked runtime type error.
- Sample verifier errors:
  - Using an uninitialized variable
  - Jumping out of code array
  - Casting an integer to a pointer
- Checks left until runtime:
  - Array bounds checking

Bytecode / Verifier Specifications

- Current specifications:
  - 30 page text description [Lindholm,Yellin]
  - Sun's reference implementation (~3500 lines of C code)
- Not concise, and vague, inconsistent areas.
- Difficult to state correctness of implementation.
- Difficult to reason about safety and security.

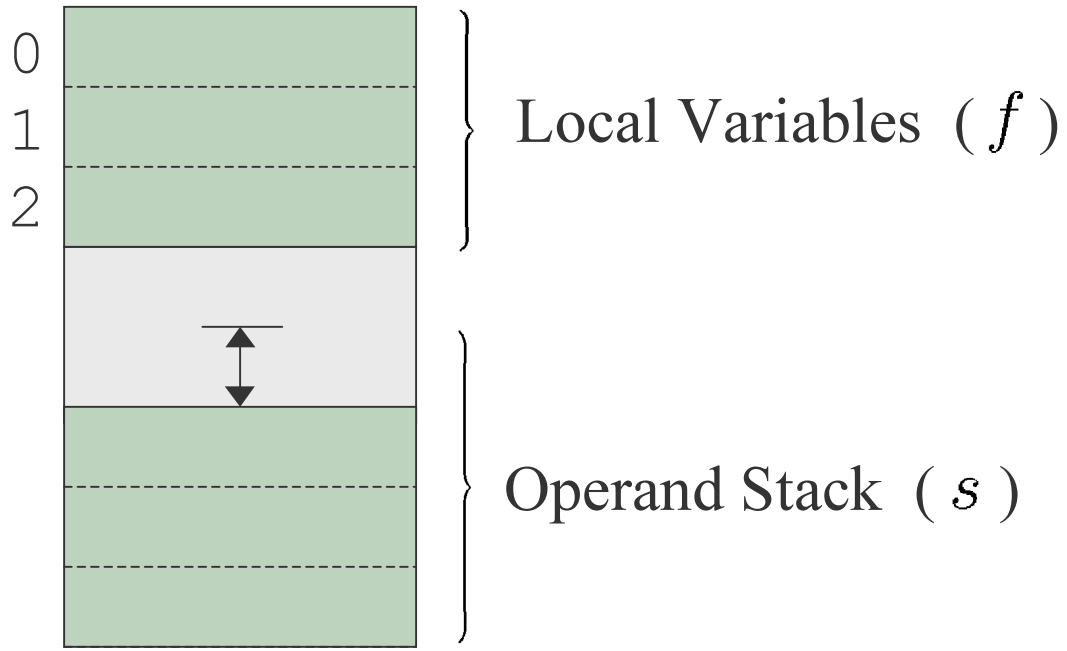
Framework

- Create JVM model.
- Describe bytecodes w/ operational semantics.
- Formulate static type rules.
- Prove soundness.
  - Well-typed programs do not cause runtime type errors when executed in the JVM model.

Programs:

- array of bytecodes
- represent bytecodes for single method

JVM Model



Program State: $\langle pc,\ f,\ s \rangle$

pc : program counter

f : VAR $\rightarrow$ value

s : stack of values

Initial State: $\langle 1,\ f_0,\ \epsilon \rangle$

JVML_i Bytecodes

- Instructions:

`push 0`	`store x`	`new` σ
`pop`	`load x`	`init` σ
`inc`	`halt`	`use` σ
`if L`		

- Operational Semantics:

$$\frac{P[pc] = \texttt{inc}}{P \vdash \langle pc,\ f,\ n \cdot s \rangle \rightarrow \langle pc+1,\ f,\ (n+1) \cdot s \rangle}$$

Runtime Errors

- Pop empty stack.
- Access illegal local variable.
- Stack values don’t match types expected by bytecode operation.
- Use uninitialized object.

Program will halt if error occurs.

Type Rules

- For program P, compute for each $i \in Dom(P)$

$$F_i : \textsc{Var} \rightarrow \texttt{type}$$
$$S_i : \texttt{stack of types}$$

- Static semantics of **`inc`**:

$$\frac{P[i] = \texttt{inc} \quad F_{i+1} = F_i \quad S_{i+1} = S_i = \textsc{Int} \cdot \alpha \quad i+1 \in Dom(P)}{F, S, i \vdash P}$$

Type Rules

- **`if`**:

$$\frac{\begin{array}{c} P[i] = \texttt{if}\ \ L \\ F_{i+1} = F_L = F_i \\ S_i = \textsc{Int} \cdot S_{i+1} = \textsc{Int} \cdot S_L \\ i+1 \in Dom(P) \\ L \in Dom(P) \end{array}}{F, S, i \vdash P}$$

- Well-typed Program:

$$\frac{\begin{array}{c} F_1 = F_{\textsc{Top}} \\ S_1 = \epsilon \\ \forall i \in Dom(P).\ F, S, i \vdash P \end{array}}{F, S \vdash P}$$

Object Initialization

- Instructions:
  - `new` σ allocate memory for object
  - `init` σ initialize object
  - `use` σ use initialized object

- **Initialization** - No object can be used before it has been initialized.

Object creation

- Example:

```
Point p = new Point(3)

1:  new Point
2:  dup
3:  iconst 3
4:  invokespecial Point.<init>(I)V
```

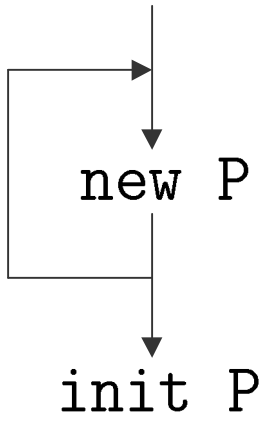
- No easy pattern to match.
- Multiple refs to same uninitialized object.

Alias Analysis

- Other situations:

```
1:   new P
2:   new P
3:   init P
```

or



- Equivalence classes based on line where object was created.

The `new` Instruction

- Uninitialized type placed on stack:

$$\frac{\begin{array}{c} P[i] = \texttt{new}\ \sigma \\ \sigma \in \texttt{T} \\ S_{i+1} = \hat{\sigma}_i \cdot S_i \\ F_{i+1} = F_i \\ \hat{\sigma}_i \notin S_i \\ \forall y \in Dom(F_i).\ F_i[y] \neq \hat{\sigma}_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$\hat{\sigma}_i$: uninitialized object of type σ allocated on line i.

The `init` Instruction

- Substitution of initialized type for uninitialized type:

$$\frac{\begin{array}{c} P[i] = \texttt{init}\ \sigma \\ \sigma \in \texttt{T} \\ S_i = \hat{\sigma}_j \cdot \alpha,\ \ j \in Dom(P) \\ S_{i+1} = [\sigma/\hat{\sigma}_j]\alpha \\ F_{i+1} = [\sigma/\hat{\sigma}_j]F_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

Initialization Invariant

- During program execution, there is never more than one value of type $\hat{\sigma}_i$ present.

- If this is violated, we could initialize one object and mistakenly believe that a different object was also initialized.

One-step Soundness

Theorem 1 (One-step Soundness) *Given* P, F, *and* S *such that* $F, S \vdash P$:

$$
\begin{aligned}
&\forall pc, f, s, pc', f', s'.\\
&\quad P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle\\
&\quad\ \wedge\ \ s : S_{pc}\\
&\quad\ \wedge\ \ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y]\\
&\quad\ \wedge\ \ ConsistentInit(s, S_{pc}, f, F_{pc})\\
&\quad \Rightarrow s' : S_{pc'}\\
&\quad\ \wedge\ \ \forall y \in \textsc{Var}.\ f'[y] : F_{pc'}[y]\\
&\quad\ \wedge\ \ ConsistentInit(s', S_{pc'}, f', F_{pc'})\\
&\quad\ \wedge\ \ pc' \in Dom(P)
\end{aligned}
$$

Progress

Theorem 2 (Progress) *Given* P, F, *and* S *such that* $F, S \vdash P$*:*

$$
\begin{aligned}
&\forall pc, f, s. \\
&\quad s : S_{pc} \\
&\quad \wedge \ \ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y] \\
&\quad \wedge \ \ ConsistentInit(s, S_{pc}, f, F_{pc}) \\
&\quad \wedge \ \ pc \in Dom(P) \\
&\quad \wedge \ \ P[pc] \neq \texttt{halt} \\
&\Rightarrow \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s \rangle \rightarrow \langle pc',\ f',\ s' \rangle
\end{aligned}
$$

Soundness

Theorem 3 (Soundness) *Given* P, F, S *such that* $F, S \vdash P$*:*

$$\forall pc, f_0, f, s.$$
$$\left(\begin{array}{l} P \vdash \langle 1,\ f_0,\ \epsilon \rangle \rightarrow^* \langle pc,\ f,\ s \rangle \\ \wedge\ \neg \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s \rangle \rightarrow \langle pc',\ f',\ s' \rangle \end{array} \right)$$
$$\Rightarrow P[pc] = \texttt{halt} \wedge s : S_{pc}$$

Extensions

- Constructors
  - constructor must call superclass constructor
- Primitive Types and Basic Operations
- Subroutines [Stata,Abadi]
  - `jsr L` allocate memory for object
  - `ret x` initialize object
  - polymorphic over untouched vars
  - class stack discipline

Bug in Sun's JDK 1.1.4

```
1:  jsr 9
2:  store 1
3:  jsr 9
4:  store 2
5:  load 2
6:  init P
7:  load 1
8:  use P
```

```
9:  store 0
10: new P
11: ret 0
```

variables 1 and 2 contain references to two different objects which both have type $\hat{P}_{10}$.

Related Work

- Java type systems
  - Java Language [Drossopoulou,Eisenbach], [Syme]
  - JVML [Stata,Abadi], [Qian]
- Concurrent constraint programs [Saraswat]
- defensive-JVM [Cohen]
- Experimental tests [Sirer,McDirmid,Bershad]
- TIL / TAL [Morrisett,Harper]

Future Work

- Additional features
  - Exceptions
  - Object system
  - Heap
  - Concurrency
- Proof methods
  - Issues of scale
  - Extensible proofs

Conclusions

- JVML_i is an effective model.
- Sound type system.
- Reason about existing implications.
- Framework is scalable.