



Static Program Analysis

Lecture 9: Dataflow Analysis VIII (Java Bytecode Verification)

Summer Semester 2018

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-18/spa/>

The Java Virtual Machine

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

The Java Virtual Machine

Java Bytecode

- **Intermediate language** between high-level language and machine code
- Execution on **Java Virtual Machine** (JVM)

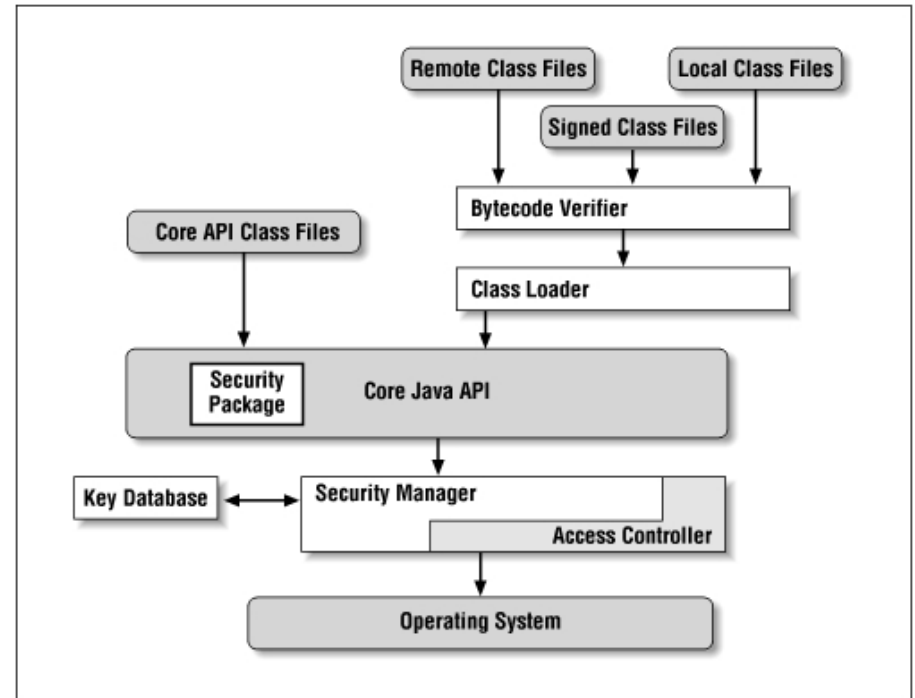
Java Bytecode

- **Intermediate language** between high-level language and machine code
- Execution on **Java Virtual Machine** (JVM)
- **Advantages:**
 - architecture independency (especially for web applications)
 - faster than pure (i.e., source code) interpretation
- **Problem: security issues**
 - destruction of data
 - modification of data
 - disclosure of personal information
 - modification of other programs

The Java Virtual Machine

Java Security: the Sandbox

- **Insulation layer** providing indirect access to system resources
- Hardware access via **API classes and methods**
- **Bytecode verification** upon loading
 - well-typedness
 - proper object referencing
 - proper control flow



The Java Virtual Machine

The Java Virtual Machine

- Conventional **stack-based abstract machine**
- Supports **object-oriented features**: classes, methods, etc.
- **Stack** for method parameters and intermediate results of expression evaluations
- **Registers** for source-level local variables
- Both part of **method activation record** (and thus preserved across method calls)
- Method entry point specifies **required number** of
 - registers (m_r)
 - stack slots (m_s ; for memory allocation)
- (Most) instructions are **typed**

The Java Virtual Machine

Example: Factorial Function

Example 9.1 (Factorial function)

Java source code:

```
static int factorial(int n)
{ int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res; }
```

The Java Virtual Machine

Example: Factorial Function

Example 9.1 (Factorial function)

Java source code:

```
static int factorial(int n)
{ int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res; }
```

Corresponding JVM bytecode:

```
method static int factorial(int), 2 registers, 2 stack slots
 1: istore 0      // store n in register 0
 2: iconst_1     // push constant 1
 3: istore 1     // store res in register 1
 4: iload 0      // push n
 5: ifle 12      // if <= 0, go to end
 6: iload 1      // push res
 7: iload 0      // push n
 8: imul        // res * n on top of stack
 9: istore 1     // store in res
10: iinc 0, -1   // decrement n
11: goto 4       // go to loop header
12: iload 1      // push res
13: ireturn     // return res to caller
```


The Java Virtual Machine

JVM Instruction Set (excerpt; \approx 200 instructions in total)

`iload n` : push integer from register n

`istore n` : pop integer into register n

`iconst_ z` : push integer z

`aconst_null`: push null reference

`iadd`: add two topmost integers on stack and push sum

`getfield C f τ` : pop reference to object o (of class C) and push value of field f of o (of type τ)

`putfield C f τ` : pop value v (of type τ) and reference to object o (of class C) and assign v to field f of o

`new C` : create new object (of class C) and push reference

`invoke C M $\tau_0(\tau_1, \dots, \tau_n)$` : pop values v_1, \dots, v_n (of type τ_1, \dots, τ_n) and reference to object o (of class C), call method M of o with parameters v_1, \dots, v_n , and push return value (of type τ_0)

`if_icmpeq l` : pop two topmost integers from stack and jump to line l if equal

`ireturn`: return to caller with integer result on top of stack

Malicious Bytecode

Example 9.2 (Malicious bytecode)

```
1:  iconst_5
2:  iconst_1
3:  putfield A f int
```

interprets second stack entry (5) as reference to object of class **A** and assigns topmost stack entry (1) to field **f** of this object

The Java Bytecode Verifier

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

The Java Bytecode Verifier

Correctness of Bytecode

Conditions to ensure **proper operation**:

Type correctness: arguments of instructions always of expected type

No stack over-/underflow: never push to full stack or pop from empty stack

Code containment: PC must always point into the method code

Register initialization: load from non-parameter register only after store

Object initialization: constructor must be invoked before using class instance

Access control: operations must respect visibility modifiers
(`private/protected/public`)

The Java Bytecode Verifier

Correctness of Bytecode

Conditions to ensure **proper operation**:

Type correctness: arguments of instructions always of expected type

No stack over-/underflow: never push to full stack or pop from empty stack

Code containment: PC must always point into the method code

Register initialization: load from non-parameter register only after store

Object initialization: constructor must be invoked before using class instance

Access control: operations must respect visibility modifiers

(`private/protected/public`)

Options:

- **dynamic checking** at execution time (“defensive JVM approach”)
 - expensive, slows down execution
- **static checking** at loading time (here)
 - verified code executable at full speed without extra dynamic checks

The Java Bytecode Verifier

The Java Bytecode Verifier

Summary: **dataflow analysis** applied to **type-level abstract interpretation** of JVM

1. Association of **type information** with register and stack contents
 - set of types forms a complete lattice
2. Simulation of **execution of instructions** at type level (“**symbolic execution**”)
3. Use **dataflow analysis** to cover all concrete executions
4. **Modular analysis**: proceeds method per method

(see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*, Journal of Automated Reasoning 30(3–4), 2003, 235–269)

The Type-Level Abstract Interpreter

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

The Type-Level Abstract Interpreter

Types

The **set of types**, *Typ*, is composed of

- **Primitive** types:
 - *int* (covering *boolean*, *byte*, *char*, *short*)
 - *long*
 - *float*
 - *double*

The Type-Level Abstract Interpreter

Types

The **set of types**, *Typ*, is composed of

- **Primitive** types:
 - *int* (covering *boolean*, *byte*, *char*, *short*)
 - *long*
 - *float*
 - *double*
- **Object reference** types: *C* for every class name *C*

The Type-Level Abstract Interpreter

Types

The **set of types**, *Typ*, is composed of

- **Primitive** types:
 - *int* (covering *boolean*, *byte*, *char*, *short*)
 - *long*
 - *float*
 - *double*
- **Object reference** types: *C* for every class name *C*
- **Array** types: $\tau[]$ for every primitive or object reference type τ

The Type-Level Abstract Interpreter

Types

The **set of types**, Typ , is composed of

- **Primitive** types:
 - int (covering $boolean$, $byte$, $char$, $short$)
 - $long$
 - $float$
 - $double$
- **Object reference** types: C for every class name C
- **Array** types: $\tau[]$ for every primitive or object reference type τ
- **Method** types: $\tau_0(\tau_1, \dots, \tau_n)$ for $n \in \mathbb{N}$, $\tau_i \in Typ$

The Type-Level Abstract Interpreter

Types

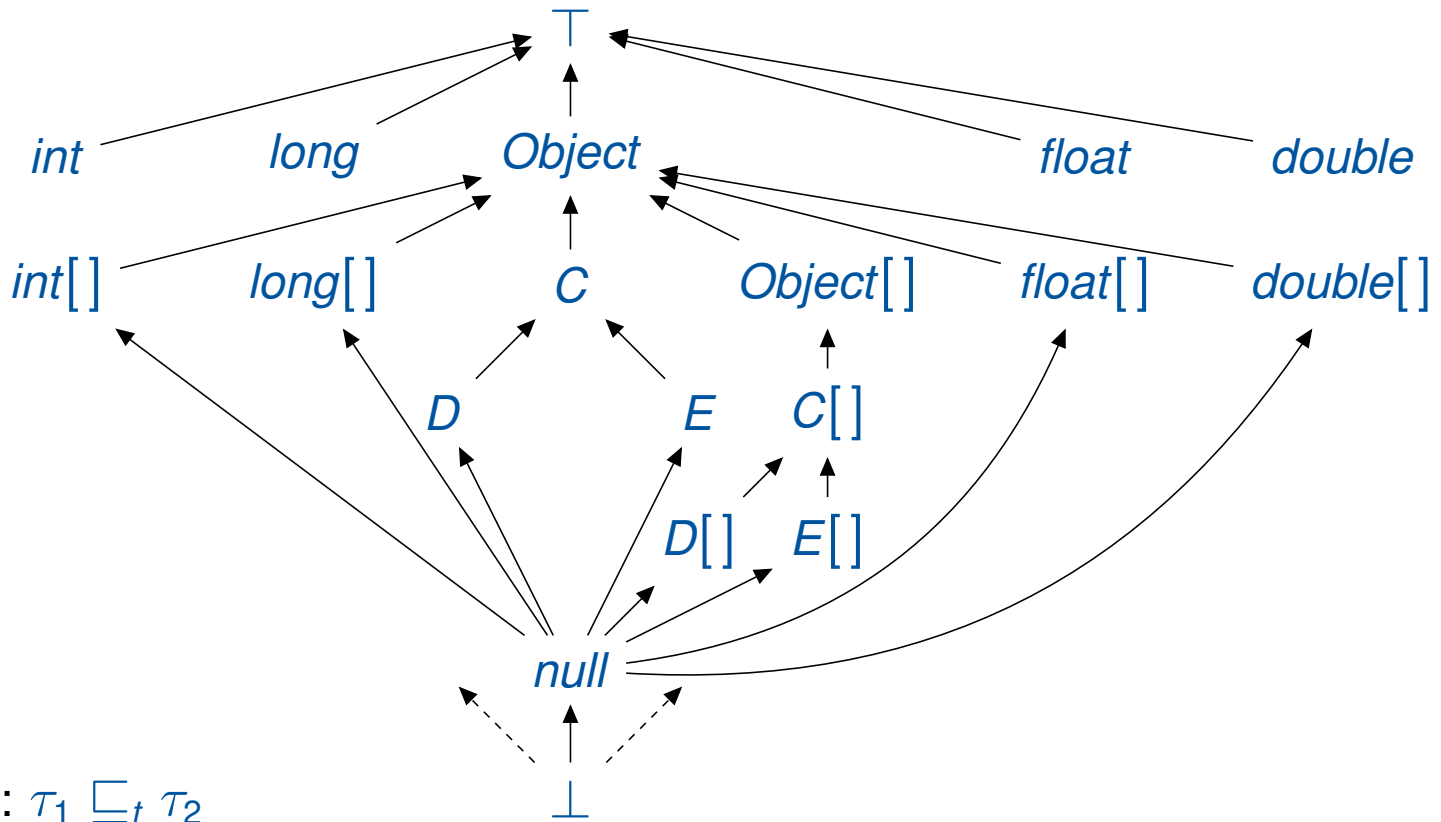
The **set of types**, Typ , is composed of

- **Primitive** types:
 - *int* (covering *boolean*, *byte*, *char*, *short*)
 - *long*
 - *float*
 - *double*
- **Object reference** types: C for every class name C
- **Array** types: $\tau[]$ for every primitive or object reference type τ
- **Method** types: $\tau_0(\tau_1, \dots, \tau_n)$ for $n \in \mathbb{N}$, $\tau_i \in Typ$
- **Special** types:
 - *null* (null reference)
 - *Object* (any object)
 - \top (contents of uninitialised registers, i.e., any value)
 - \perp (absence of any value)

The Type-Level Abstract Interpreter

The Subtyping Relation (excerpt)

(C, D, E user-defined classes; D, E extending C)



Notation: $\tau_1 \sqsubseteq_t \tau_2$

The Type-Level Abstract Interpreter

The Type-Level Abstract Interpreter

- **Idea:** execute JVM instructions on **types** (rather than concrete values)
 - **stack type** $S \in \text{Typ}^{\leq m_s}$ (top to the left)
 - **register type** $R : \{0, \dots, m_r - 1\} \rightarrow \text{Typ}$

The Type-Level Abstract Interpreter

The Type-Level Abstract Interpreter

- **Idea:** execute JVM instructions on **types** (rather than concrete values)
 - **stack type** $S \in \text{Typ}^{\leq m_s}$ (top to the left)
 - **register type** $R : \{0, \dots, m_r - 1\} \rightarrow \text{Typ}$
- Represented as **transition relation**

$$i : (S, R) \triangleright (S', R')$$

where

- i : current instruction
- (S, R) : stack/register type before execution
- (S', R') : stack/register type after execution

The Type-Level Abstract Interpreter

The Type-Level Abstract Interpreter

- **Idea:** execute JVM instructions on **types** (rather than concrete values)
 - **stack type** $S \in \text{Typ}^{\leq m_s}$ (top to the left)
 - **register type** $R : \{0, \dots, m_r - 1\} \rightarrow \text{Typ}$
- Represented as **transition relation**

$$i : (S, R) \triangleright (S', R')$$

where

- i : current instruction
- (S, R) : stack/register type before execution
- (S', R') : stack/register type after execution
- **Errors** (type mismatch, stack over-/underflow, ...) denoted by absence of transition

The Type-Level Abstract Interpreter

Some Transition Rules

<code>iconst_z :</code>	$(S, R) \triangleright (int.S, R)$	if $ S < m_s$
<code>aconst_null :</code>	$(S, R) \triangleright (null.S, R)$	if $ S < m_s$
<code>iadd :</code>	$(int.int.S, R) \triangleright (int.S, R)$	
<code>if_icmpeq l :</code>	$(int.int.S, R) \triangleright (S, R)$	
<code>iload n :</code>	$(S, R) \triangleright (int.S, R)$	if $0 \leq n < m_r, R(n) = int, S < m_s$
<code>aload n :</code>	$(S, R) \triangleright (R(n).S, R)$	if $0 \leq n < m_r, R(n) \sqsubseteq_t Object, S < m_s$
<code>istore n :</code>	$(int.S, R) \triangleright (S, R[n \mapsto int])$	if $0 \leq n < m_r$
<code>astore n :</code>	$(\tau.S, R) \triangleright (S, R[n \mapsto \tau])$	if $0 \leq n < m_r, \tau \sqsubseteq_t Object$
<code>getfield C f τ :</code>	$(D.S, R) \triangleright (\tau.S, R)$	if $D \sqsubseteq_t C$
<code>putfield C f τ :</code>	$(\tau'.D.S, R) \triangleright (S, R)$	if $\tau' \sqsubseteq_t \tau, D \sqsubseteq_t C$
<code>invoke C M σ :</code>	$(\tau'_n \dots \tau'_1.\tau'.S, R) \triangleright (\tau_0.S, R)$	if $\sigma = \tau_0(\tau_1, \dots, \tau_n), \tau'_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n, \tau' \sqsubseteq_t C$

Some Theoretical Properties

Lemma 9.3

1. (Typ, \sqsubseteq_t) is a *complete lattice satisfying ACC*.

Some Theoretical Properties

Lemma 9.3

1. (Typ, \sqsubseteq_t) is a *complete lattice satisfying ACC*.
2. (*Determinacy*) The transitions of the abstract interpreter define a partial function:
If $i : (S, R) \triangleright (S_1, R_1)$ and $i : (S, R) \triangleright (S_2, R_2)$, then $S_1 = S_2$ and $R_1 = R_2$.

The Type-Level Abstract Interpreter

Some Theoretical Properties

Lemma 9.3

1. (Typ, \sqsubseteq_t) is a *complete lattice satisfying ACC*.
2. (*Determinacy*) The transitions of the abstract interpreter define a partial function:
If $i : (S, R) \triangleright (S_1, R_1)$ and $i : (S, R) \triangleright (S_2, R_2)$, then $S_1 = S_2$ and $R_1 = R_2$.
3. (*Soundness*) If $i : (S, R) \triangleright (S', R')$, then for all concrete states (s, r) matching (S, R) , the defensive JVM will not stop with a run-time type exception when applying i to (s, r) (but rather change to some (s', r') matching (S', R')).

Proof.

see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations* □

The Dataflow Analysis

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

The Dataflow Analysis

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$

- **Extremal label** $E := \{1\}$ (forward problem)

- **Flow relation** F : for every $l \in Lab$,

$$\left\{ \begin{array}{ll} (l, m), (l, l+1) \in F & \text{if } l: \text{ conditional jump to } m \\ (l, m) \in F & \text{if } l: \text{ unconditional jump to } m \\ - & \text{if } l: \text{ return instruction} \\ (l, l+1) & \text{otherwise} \end{array} \right.$$

The Dataflow Analysis

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$
- **Extremal label** $E := \{1\}$ (forward problem)
- **Flow relation** F : for every $l \in Lab$,

$$\begin{cases} (l, m), (l, l+1) \in F & \text{if } l: \text{ conditional jump to } m \\ (l, m) \in F & \text{if } l: \text{ unconditional jump to } m \\ - & \text{if } l: \text{ return instruction} \\ (l, l+1) & \text{otherwise} \end{cases}$$

- **Complete lattice** (D, \sqsubseteq) where

$$- D := \underbrace{Typ^{\leq m_s}}_{\text{stack}} \times \underbrace{\{0, \dots, m_r - 1\}}_{\text{registers}} \triangleright Typ \cup \{ \underbrace{None}_{\text{least element}}, \underbrace{Error}_{\text{untypeable}} \}$$

- for every $(S, R) \in D$, $None \sqsubseteq (S, R)$ and $(S, R) \sqsubseteq Error$

- $(S_1, R_1) \sqsubseteq (S_2, R_2)$ iff

- $S_1 = \sigma_1 \dots \sigma_n$, $S_2 = \tau_1 \dots \tau_n$ (same length $n \in \mathbb{N}!$), $\sigma_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n$
- $R_1(i) \sqsubseteq_t R_2(i)$ for $0 \leq i < m_r$

The Dataflow Analysis

The Dataflow System II

- **Extremal value** (for parameter types τ_1, \dots, τ_n of M):

$$\iota := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

The Dataflow Analysis

The Dataflow System II

- **Extremal value** (for parameter types τ_1, \dots, τ_n of M):

$$\iota := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

- **Transfer functions** $\{\varphi_l \mid l \in Lab\}$:

$$\varphi_l(S, R) := \begin{cases} (S', R') & \text{if } l : i \text{ and } i : (S, R) \triangleright (S', R') \\ Error & \text{otherwise} \end{cases}$$

The Dataflow Analysis

The Dataflow System II

- **Extremal value** (for parameter types τ_1, \dots, τ_n of M):

$$\iota := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

- **Transfer functions** $\{\varphi_l \mid l \in \text{Lab}\}$:

$$\varphi_l(S, R) := \begin{cases} (S', R') & \text{if } l : i \text{ and } i : (S, R) \triangleright (S', R') \\ \text{Error} & \text{otherwise} \end{cases}$$

Monotonicity of transfer functions is ensured by the following lemma.

Lemma 9.4

If $i : (S, R) \triangleright (S', R')$ and $(S_1, R_1) \sqsubseteq (S, R)$, then there exists $(S'_1, R'_1) \in D$ such that $i : (S_1, R_1) \triangleright (S'_1, R'_1)$ and $(S'_1, R'_1) \sqsubseteq (S', R')$.

Proof.

see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations* □

Examples of Bytecode Verification

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	<i>None</i>
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	<i>None</i>
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	<i>None</i>
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	<i>None</i>
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	<i>None</i>
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	<i>None</i>
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	<i>None</i>
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	<i>None</i>
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	<i>None</i>
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	(int, C)
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	<i>None</i>
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	(int, C)
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	$(int.int, C)$
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	<i>None</i>
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	(int, C)
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	$(int.int, C)$
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	<i>None</i>

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	(int, C)
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	$(int.int, C)$
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	(C, C)

Examples of Bytecode Verification

Example of Correct Bytecode

Example 9.5

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at block entry)
1	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(B, \top)
2	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, B)
3	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(B, B)
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \triangleright (C.S, R)$	$(int.B, B)$
5	<code>astore 0</code>	$(\tau.S, R) \triangleright (S, R[0 \mapsto \tau])$	(C, B)
6	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
7	<code>getfield C f int</code>	$(C.S, R) \triangleright (int.S, R)$	(C, C)
8	<code>iconst_0</code>	$(S, R) \triangleright (int.S, R)$	(int, C)
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \triangleright (S, R)$	$(int.int, C)$
10	<code>aload 0</code>	$(S, R) \triangleright (R(0).S, R)$	(ε, C)
11	<code>areturn</code>	$(\tau.S, R) \triangleright (\tau.S, R)$	(C, C)

- Thus: no type errors, expected return type `(C)`

Examples of Bytecode Verification

Example of Malicious Bytecode

Example 9.6 (cf. Example 9.2)

- Assumption: class `A` provides field `f` of type `int`
- Program interprets second stack entry (5) as reference to `A`-object and assigns first stack entry (1) to field `f`
- $m_s = 2, m_r = 0$
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R) (at block entry)
<code>l</code>	<code>...</code>	<code>...</code>	<code>...</code>
<code>l + 1</code>	<code>iconst_5</code>	$(S, R) \triangleright (int.S, R)$	$(\varepsilon, \varepsilon)$
<code>l + 2</code>	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	<code>None</code>
<code>l + 3</code>	<code>putfield A f int</code>	$(int.A.S, R) \triangleright (S, R)$	<code>None</code>
<code>l + 4</code>	<code>...</code>	<code>...</code>	<code>None</code>

Examples of Bytecode Verification

Example of Malicious Bytecode

Example 9.6 (cf. Example 9.2)

- Assumption: class *A* provides field *f* of type *int*
- Program interprets second stack entry (5) as reference to *A*-object and assigns first stack entry (1) to field *f*
- $m_s = 2, m_r = 0$
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R) (at block entry)
<i>l</i>
<i>l</i> + 1	<code>iconst_5</code>	$(S, R) \triangleright (int.S, R)$	$(\varepsilon, \varepsilon)$
<i>l</i> + 2	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(int, ε)
<i>l</i> + 3	<code>putfield A f int</code>	$(int.A.S, R) \triangleright (S, R)$	<i>None</i>
<i>l</i> + 4	<i>None</i>

Examples of Bytecode Verification

Example of Malicious Bytecode

Example 9.6 (cf. Example 9.2)

- Assumption: class *A* provides field *f* of type *int*
- Program interprets second stack entry (5) as reference to *A*-object and assigns first stack entry (1) to field *f*
- $m_s = 2, m_r = 0$
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R) (at block entry)
<i>l</i>
<i>l</i> + 1	<code>iconst_5</code>	$(S, R) \triangleright (int.S, R)$	$(\varepsilon, \varepsilon)$
<i>l</i> + 2	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(int, ε)
<i>l</i> + 3	<code>putfield A f int</code>	$(int.A.S, R) \triangleright (S, R)$	$(int.int, \varepsilon)$
<i>l</i> + 4	None

Examples of Bytecode Verification

Example of Malicious Bytecode

Example 9.6 (cf. Example 9.2)

- Assumption: class `A` provides field `f` of type `int`
- Program interprets second stack entry (5) as reference to `A`-object and assigns first stack entry (1) to field `f`
- $m_s = 2, m_r = 0$
- Application of worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R) (at block entry)
<code>l</code>	<code>...</code>	<code>...</code>	<code>...</code>
<code>l + 1</code>	<code>iconst_5</code>	$(S, R) \triangleright (int.S, R)$	$(\varepsilon, \varepsilon)$
<code>l + 2</code>	<code>iconst_1</code>	$(S, R) \triangleright (int.S, R)$	(int, ε)
<code>l + 3</code>	<code>putfield A f int</code>	$(int.A.S, R) \triangleright (S, R)$	$(int.int, \varepsilon)$
<code>l + 4</code>	<code>...</code>	<code>...</code>	<i>Error</i>

Examples of Bytecode Verification

Soundness of Bytecode Verifier

Theorem 9.7

If dataflow analysis yields $AI_l \neq \text{Error}$ for every $l \in \text{Lab}$, then the analysed method *will not stop with a run-time type exception* when run on the JVM. Here run-time type exceptions refer to

- using instruction operands of wrong type (“Expecting to find ... on stack”),
- method return values of wrong type (“Wrong return value”),
- type-incompatible assignments to fields (“Incompatible type for setting field”),
- different stack sizes at the same location (“Inconsistent stack height”),
- stack overflows (i.e., more than m_s entries) (“Stack size too large”), and
- stack underflows (i.e., pop from empty stack) (“Unable to pop operand off an empty stack”).

Further Issues in Bytecode Verification

Outline of Lecture 9

The Java Virtual Machine

The Java Bytecode Verifier

The Type-Level Abstract Interpreter

The Dataflow Analysis

Examples of Bytecode Verification

Further Issues in Bytecode Verification

Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up transfer functions for **sequences of instructions** (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up transfer functions for **sequences of instructions** (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Example 9.8 (cf. Example 9.1)

```
method static int factorial(int), 2 registers, 2 stack slots
  1: istore 0      // store n in register 0
  2: iconst_1     // push constant 1
  3: istore 1     // store res in register 1
  4: iload 0      // push register n
  5: ifle 12      // if <= 0, go to end
  6: iload 1      // push res
  7: iload 0      // push n
  8: imul         // res * n on top of stack
  9: istore 1     // store in res
 10: iinc 0, -1   // decrement n
 11: goto 4      // go to loop header
 12: iload 1      // push res
 13: ireturn     // return res to caller
```

(12 instructions)

Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up transfer functions for **sequences of instructions** (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Example 9.8 (cf. Example 9.1)

```
method static int factorial(int), 2 registers, 2 stack slots
  1: istore 0      // store n in register 0
  2: iconst_1     // push constant 1
  3: istore 1     // store res in register 1
  4: iload 0      // push register n
  5: ifle 12      // if <= 0, go to end
  6: iload 1      // push res
  7: iload 0      // push n
  8: imul        // res * n on top of stack
  9: istore 1     // store in res
 10: iinc 0, -1   // decrement n
 11: goto 4      // go to loop header
 12: iload 1     // push res
 13: ireturn     // return res to caller
```

(4 extended basic blocks)

Further Issues in Bytecode Verification

Bytecode Verification on Small Devices

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems
(mobile phones, smart cards, PDAs, ...)

Further Issues in Bytecode Verification

Bytecode Verification on Small Devices

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)
- **Example: Java SmartCard**
 - 8-bit microprocessor
 - ≤ 5 kB RAM (volatile, fast)
 - ≤ 256 kB EEPROM (persistent, slow)
 - ≤ 256 kB ROM (operating system)
 - ⇒ RAM too small to store dataflow infos

Further Issues in Bytecode Verification

Bytecode Verification on Small Devices

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)
- **Example: Java SmartCard**
 - 8-bit microprocessor
 - ≤ 5 kB RAM (volatile, fast)
 - ≤ 256 kB EEPROM (persistent, slow)
 - ≤ 256 kB ROM (operating system)
 - ⇒ RAM too small to store dataflow infos
- **Solutions:**
 - Use **EEPROM** to hold verifier data structures (slow)
 - **Off-card verification** using **certificates** (see following slides)
 - **On-card verification** with **off-card code transformation** (see following slides)

Further Issues in Bytecode Verification

Off-Card Verification Using Certificates

(aka “lightweight bytecode verification using certificates”)

- Inspired by “**proof-carrying code approach**”
- Bytecode producer **attaches type information** to bytecode (“certificates”)
- Embedded system **checks well-typedness** of code (rather than inferring types)
- Advantages:
 - type checking **faster** than inference (no fixpoint iteration)
 - only **reading access** to certificates \implies can be kept in EEPROM
- Practical limitation: certificates require \approx **50% of size of annotated code**
- Implementation: **Sun’s K Virtual Machine** (KVM)

Further Issues in Bytecode Verification

On-Card Verification with Off-Card Transformation

- Standard bytecode verification (solving dataflow equations using fixpoint iteration) on **normalized bytecode**
- Bytecode restrictions:
 - organised as **extended basic blocks** (cf. Slide 9.26)
 - only **one register type** shared by all control points (= entry points of extended basic blocks)
 - **stack empty** before each jump target and after each jump instruction
(= entry/exit points of extended basic blocks)
- **Space complexity** of bytecode verification ($|Lab|/m_s/m_r$ = number of blocks/stack entries/registers):
 - without restriction: $\mathcal{O}(|Lab| \cdot (m_s + m_r))$
 - with restriction: $\mathcal{O}(m_s + m_r)$
 - m_s : for stack type analysis of single extended basic block
 - m_r : for global register type
- Restrictions ensured by off-card (i.e., compile-time) **code transformation**
 - stack normalizations around jumps
 - register re-allocation by graph coloring
 - can increase code size and number of used registers (but negligible on “typical” Java Card code)

Further Issues in Bytecode Verification

Reducing the Instruction Set

- Another obstacle in formalisation of Java bytecode: \approx 200 instructions

Further Issues in Bytecode Verification

Reducing the Instruction Set

- Another obstacle in formalisation of Java bytecode: \approx 200 instructions
- **Approach:** reduction to “representative” set of instructions

Further Issues in Bytecode Verification

Reducing the Instruction Set

- Another obstacle in formalisation of Java bytecode: \approx 200 instructions
- **Approach:** reduction to “representative” set of instructions
- **Examples:**
 - Jimple (Soot framework, 15 operations)
 - BIR (Sawja framework, 11 operations)

Reducing the Instruction Set

- Another obstacle in formalisation of Java bytecode: \approx 200 instructions
- **Approach:** reduction to “representative” set of instructions
- **Examples:**
 - Jimple (Soot framework, 15 operations)
 - BIR (Sawja framework, 11 operations)
- Systematic investigation:

J. Chrzaszcz, P. Czarnik, A. Schubert: *A Dozen Instructions Make Java Bytecode*,
Electronic Notes in Theoretical Computer Science 264(4), 2011, 19–34