Static Program Analysis

Lecture 11: Dataflow Analysis X (Java Bytecode Verification)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/spa11/

Summer Semester 2011

Outline

1. Repetition: The Java Bytecode Verifier
2. Examples of Bytecode Verification
3. Further Issues in Bytecode Verification

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

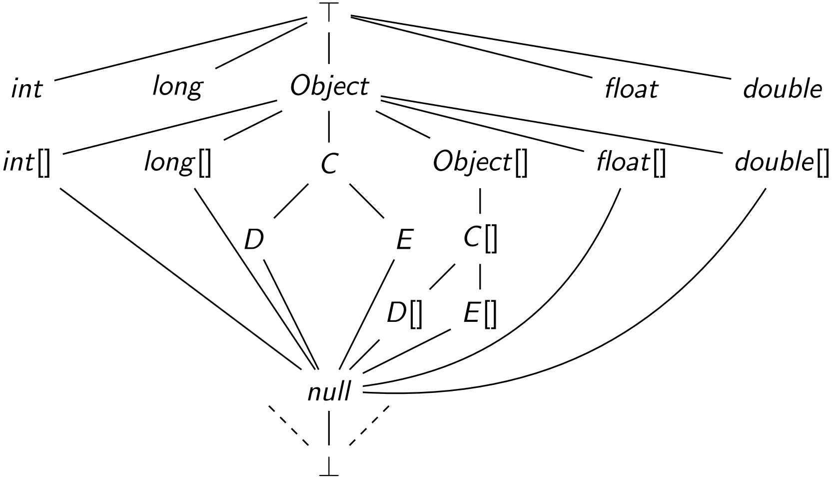
Summary: dataflow analysis applied to type-level abstract interpretation of JVM

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
3. Use dataflow analysis to cover all concrete executions
4. 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 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 I

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

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

  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 II

Some transition rules:

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

The Dataflow System I

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

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

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

- Complete lattice $(D, \sqsubseteq)$ where
  - $D := \underbrace{Typ^{\leq m_s}}_{\text{stack}} \times \underbrace{\{0, \ldots, m_r - 1\} \to Typ}_{\text{registers}} \cup \{ \underbrace{None}_{\text{least element}}, \underbrace{Error}_{\text{untypeable}} \}$
  - 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
    - $S_1 = \sigma_1 \ldots \sigma_n$, $S_2 = \tau_1 \ldots \tau_n$ (same length!), $\sigma_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n$
    - $R_1(i) \sqsubseteq_t R_2(i)$ for $0 \leq i < m_r$

The Dataflow System II

- Extremal value

$$\iota := (\varepsilon, (\tau_1, \ldots, \tau_n, \underbrace{\top, \ldots, \top}_{m_r - n \text{ times}}))$$

  with parameter types $\tau_1, \ldots, \tau_n$ of M
- Transfer functions $\{\varphi_l \mid l \in L\}$ are defined by

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

Monotonicity of transfer functions is ensured by the following lemma.

Lemma

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

Proof.

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

Outline

Example of Correct Bytecode

Example 11.1

- 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: on the board

Label	Instruction	Transition rule (w/o conditions)
1	`aload 0`	$(S, R) \to (R(0).S, R)$
2	`iconst_1`	$(S, R) \to (int.S, R)$
3	`invoke B M C(int)`	$(int.\mathtt{B}.S, R) \to (\mathtt{C}.S, R)$
4	`astore 0`	$(\tau.S, R) \to (S, R[0 \mapsto \tau])$
5	`aload 0`	$(S, R) \to (R(0).S, R)$
6	`getfield C f int`	$(\mathtt{C}.S, R) \to (int.S, R)$
7	`iconst_0`	$(S, R) \to (int.S, R)$
8	`if_icmpeq 1`	$(int.int.S, R) \to (S, R)$
9	`aload 0`	$(S, R) \to (R(0).S, R)$
10	`areturn`	$(S, R) \to (S, R)$

Example of Malicious Bytecode

Example 11.2 (cf. Example 10.2)

- 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: on the board

Label	Instruction	Transition rule (w/o conditions)
1	`iconst_5`	$(S, R) \to (int.S, R)$
2	`iconst_1`	$(S, R) \to (int.S, R)$
3	`putfield A f int`	$(int.\mathtt{A}.S, R) \to (S, R)$
4	...	

Soundness of Bytecode Verifier

Theorem 11.3

If dataflow analysis yields $\mathrm{AI}_l \neq Error$ for every $l \in L$, then the analyzed 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`*"*).

Moreover it is excluded that different stack sizes occur at the same location (*"*`Inconsistent stack height`*"*).

Outline

1. Repetition: The Java Bytecode Verifier
2. Examples of Bytecode Verification
3. Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up dataflow equations only for extended basic blocks (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 11.4 (cf. Example 10.1)

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

(12 instructions) (4 extended basic blocks)

Bytecode Verification on Small Devices

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

- **Problem:** bytecode verification is expensive
  $\Longrightarrow$ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)
- **Example:** Java SmartCard
  - 8-bit microprocessor
  - $\approx$ 2 kB RAM (volatile, fast)
  - $\approx$ 80 kB EEPROM (persistent, slow)
  - $\approx$ 100 kB ROM (operating system)

  $\Longrightarrow$ 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)

Off-Card Verification Using Certificates

(also: "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 $\Longrightarrow$ can be kept in EEPROM
- Practical limitation: certificates require $\approx$ 50% of size of annotated code
- Implementation: Sun's K Virtual Machine (KVM)

On-Card Verification with Off-Card Transformation

- Standard bytecode verification (solving dataflow equations using fixpoint iteration) on normalized bytecode
- Bytecode restrictions:
  - 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 ($|L|/m_s/m_r$ = number of blocks/stack entries/registers):
  - w/o restriction: $\mathcal{O}(|L| \cdot (m_s + m_r))$
  - with restriction: $\mathcal{O}(m_s + m_r)$
- 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)