Java™ Virtual Machine Byte Code Verification: Past, Present and Future

Gilad Bracha
Computational Theologist
Sun Microsystems

Introduction

Byte code verification:

The process of inferring valid types for Java virtual machine language

Verification often confused with other, separate safety checks performed by the JVM (e.g., format checking, access control)

The Basic Idea of BCV

By ensuring the type safety of JVML code

"statically" at link time, we avoid the need for

potentially costly dynamic checks

(e.g., are the operands of an **iadd** instruction really integers?)

while preserving both security

and the integrity of the VM

A Bit of History

- Java was originally targeted at circa 1992 cable TV set top boxes
- Interactive television was the "next big thing"
- Security was not a big deal: code would be distributed over a a closed, proprietary network
- No need for verification

Verification as Afterthought

- Interactive TV didn't happen as fast as anticipated
- Someone realized this could be retargeted for the internet
- Security becomes a real concern
- Enter verification

The Need for Inference

- Class file format does not carry sufficient type information
- In particular, no type information for Local Variables, Operand Stack
- This information can be inferred.

Advantages of Type Inference

- Reduced size requirements for class files:
  - Saves bandwidth (most precious resource for applets in mid-90s)
- Minimal changes to format

Disadvantages of Type Inference

- Complexity
- Speed
- Memory consumption

How Verification Works (1)

- Maintain a work queue of instructions
- Maintain a table associating instructions and incoming type states
- Initially, instruction 0 associated with initial type state
  - this in L0, arguments in L1 .. Lk, other locals undefined
  - empty operand stack.
- All other instructions have no type state associated with them initially.

How Verification Works (2)

```
Put instruction 0 on work queue

While queue not empty {

   Simulate instruction based on associated incoming
    type state to derive outgoing type state

   For each successor instruction si {

        place next instruction on queue

        If si has an associated type state, merge
         outgoing type state with si's state.

         If result of merge differs from si's
          recorded type state, update type state for
          si, and place si on work queue

}
```


How Verification Works (3)

- If not unconditional branch, successor instructions include next instruction
- If branch, successor instructions include target
- Successors also include any applicable exception handlers (their type state is special, as operand stack will contain only exception)

A Simple Example (1)

```
int foo(boolean p) {

  int i;
  float f;

  if (p) {
    i = 6;
  }
  else {
    f = 2.0
  }
  return i; // illegal program - i is not
   definitely assigned
}
```


A Simple Example (2)

```
   iload 1
   ifeq L1
   iconst 6
   istore 2
   goto L2
L1:fconst 2.0
   fstore 3
L2:iload 2 // will not verify - local 2 not
           // guaranteed to have type int
   ireturn
```

Type inference strategy imposes requirements at language level; the two must match (they haven't always, so we have had programs that are legal Java but will not verify)

Complications

- Subroutines
- Object initialization

Subroutines (1)

Motivation: **`try-finally`**

Typical pattern:

```
try {
 doSomeThingThatMightFail();
}
catch (expectedException e)
  {callHandler(e);}
finally { cleanUp();}
```

Must ensure clean up gets done in both normal and exceptional cases

Subroutines (2)

```
    aload 0
    invokevirtual doSomethingThatMightFail
    jsr H2
    return
H1:aload 0
    invokevirtual callHandler
    jsr H2
    return
H2:astore 1
    aload 0
    invokevirtual cleanup
    ret 1
```

Subroutine prevents duplication of cleanup code

Subroutines (3): Polymorphism

```
    aload 0
    iconst 7
    dup
    istore 2
    jsr H2
    ireturn
    getstatic C.P
    astore 2
H1: aload 0
    invokevirtual callHandler
    jsr H2
    return
H2: astore 1
    aload 0
    invokevirtual cleanup
    ret 1
```

Subroutines (4)

- Local variable **L2** has different types on different control paths
- The subroutine doesn't care about **L2**, yet straightforward inference will fail.

So, we try and be smart. Complicates the algorithm quite a bit.

Subroutines (5): Exact GC

- Local variable **L2** has different types on different control paths!
- May be a pointer on one path and an **int** on another.
- GC needs to maintain pointer maps, but at ***H2*** type of **L2** is ambiguous.
- GC must split **L2** into two distinct variables.
- What was the point of sharing **L2** in the first place?

Subroutines (6): Summary

- Locals cannot be shared with accurate GC anyway
- Studies show code space savings negligible
- Premature Optimization

Disadvantages of Complexity

- Hard to prove correctness
- Hard to maintain
- Hard to replicate and adapt (e.g., laziness, shared code)

Performance Disadvantages

- Startup time
  - hack: do not verify system code
    - Javac bugs detected late
    - JIT works harder
- Footprint
  - Javacard uses different solution
  - J2ME uses yet another solution (more on this below)

From Type Inference to Type Checking

- Add type information for local vars and operand stack
- Classfile space penalty 5-10%
- Footprint radically reduced
  - Faster
  - Simpler
- Premature optimization is the root of all evil

JVML Typechecking

- Now the standard for J2ME CLDC
- We hope to adopt in JDK1.5 (subject to JCP)

How Typechecking Works (1)

- Associate declared type state with select instructions
- Iterate thru instructions, starting at instruction 0 with initial type state as incoming type state

How Typechecking Works (2)

Is there a declared type state?

If so, is incoming type state a subtype of it?

If not, error

else use declared type state instead

- Simulate instruction, deriving outgoing type state
- Use outgoing type state as incoming type state for next instruction.

How Typechecking Works (3)

- If branch, ensure target has declaration
- If conditional branch, ensure that computed type state is subtype of declaration at targe

Current Status

- CLDC uses typechecking rather than inference
- Executable Prolog Specification of Typechecking in Progress
- We plan to propose using typechecking in J2SE/EE in JDK1.5

End

Summary

- Type Inference and subroutines were both premature optimizations
- Happy ending: faster, smaller, more secure, more portable verifier in Java's future

