Java byte code: semantics and verification

PAS

M2R Informatique, U. Rennes 1

Thomas Jensen

Plan

1. Operational semantics of Java byte code

Plan

1. Operational semantics of Java byte code
2. Java byte code verification

Semantics of Java byte code

An example of an operational semantics in the form of an abstract machine (the Java virtual machine).

References:

- Tim Lindholm , Frank Yellin, Gilad Bracha, Alex Buckley. Java Virtual Machine Specification, Java SE 8 Edition, 2015.
- S. Freund , J. Mitchell : A Type System for the Java Bytecode Language and Verifier, Journal of Automated Reasoning, Volume 30, Issue 3-4, Pages: 271 - 321, (2003)

Java

Java: a class-based, object-oriented programming language.

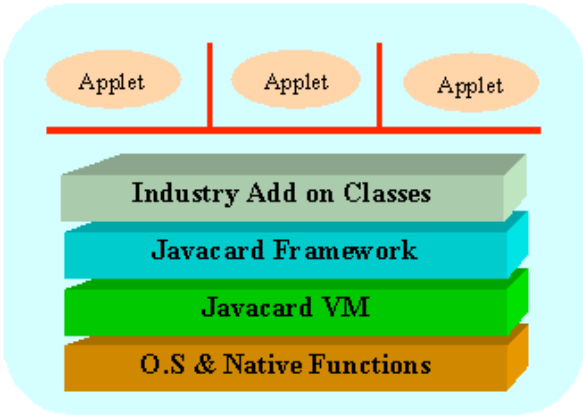
```
public class Bicycle{

    private int gear;

    private int id;

    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        id = ++numberOfBicycles;}

     public void setGear(int newValue){
        gear = newValue;}

public class MountainBike extends Bicycle {

    // the MountainBike subclass has one field
    public int seatHeight;
    ... }
```

Java byte code

To enhance portability, Java defines a byte code language that is interpreted by the Java virtual machine.

The JVM is a stack machine with registers.

```
    ...
    7 : ipush 0
// push constant 0 on stack
    8 : iload 2
// load value from register 2 onto stack
    9 : iload 1
    10 : isub
// subtract two top elements of the stack
    11 : if_icmpge 56
// compare and jump to 56...
    ...
```



Java byte code: factorial

Source code

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

Byte code

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 // push the integer constant 1
1: istore 1 // store it in register 1 (the res variable)
2: iload 0 // push register 0 (the n parameter)
3: ifle 14 // if negative or null, go to PC 14
6: iload 1 // push register 1 (res)
7: iload 0 // push register 0 (n)
8: imul // multiply the two integers at top of stack
9: istore 1 // pop result and store it in register 1
10: iinc 0, -1 // decrement register 0 (n) by 1
11: goto 2 // go to PC 2
14: iload 1 // load register 1 (res)
15: ireturn // return its value to caller
```

The Java virtual machine state

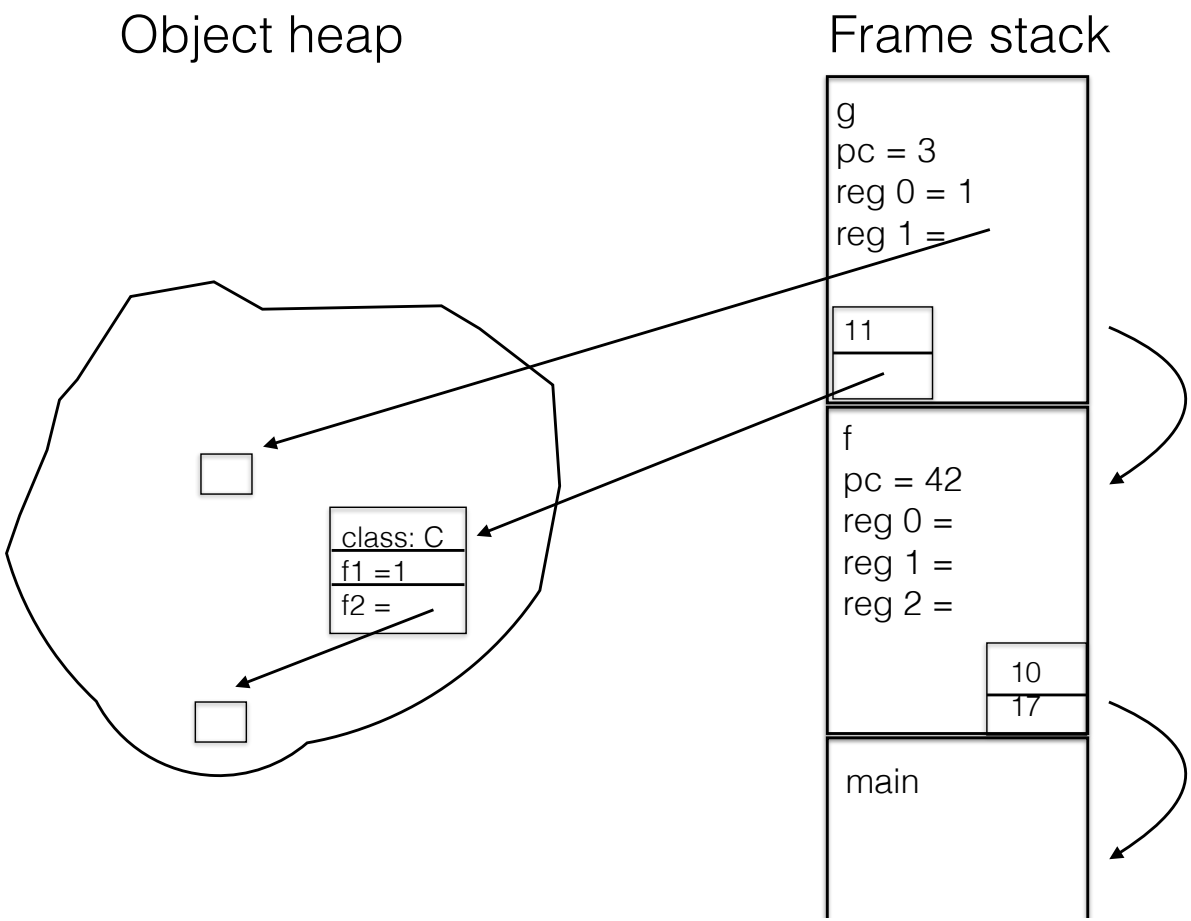
The Java virtual machine has several components:

A code component that contains the byte code of all methods of all classes loaded into the virtual machine. Our machine will have a program counter *pc* pointing to the next instruction to execute.

A heap (or memory) of objects that have been created during execution of the program.

A frame stack of methods under execution. Each frame contains an operand stack and variables local to a method call.

Heap and frame stack



The Java virtual machine specification

iadd

Operation

Add `int`

Format

iadd

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* ⇒ ..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a runtime exception.

Idealized byte code syntax

Instruction ::= `nop`

`const` c	stack manipulation
`pop`	
`dup`	
`dup2`	
`swap`	
`numop` op	
`load` x	local variables manipulation
`store` x	

Some bytecodes are **typed** (`iconst`, `aload`) to indicate whether they operate on integers, references, arrays, etc.

We leave out the types here, and put them in again when considering byte code verification.

Idealized byte code syntax (continued)

More instructions:

`ifle` pc	jump
`goto` pc	
`new` cl	heap manipulation
`putfield` f	
`getfield` f	

Real Java byte code contains many more "if"s: `if_eq`, `if_ge`, `if_nonnull`, `if_acmpeq`,...

The field in a `putfield` f instruction carries extra typing information of the form $C : f : t$ where

- C is the class declaring the field,
- t is the type of the field.

Idealized byte code syntax (continued)

More instructions

$$\left.\begin{array}{l}\texttt{invokevirtual}\ m_{id}\\ \texttt{return}\end{array}\right\}\ \text{method call and return}$$

The method identifier m_{id} is of form

$$C : m : sig$$

where C is the class of declaration and sig the method signature.

Not included here: `invokespecial` and `invokeinterface`.

Program structure

A **program** P is a collection of classes, organized into a tree-structured class hierarchy, denoted by classes(P).

To each **class** is associated

- a class name nameClass(c),
- a set of methods defined in the class.

A **method** consists of

- its code (sequence of byte codes),
- a signature that specifies its argument types and return type,
- max stack size and max number of registers used.

Lookup(M, cl) :

> *locate the most recent (re-)definition of method M starting from class cl and going upwards in the class hierarchy.*

Operational semantics

Semantic domains.

$$
\begin{array}{rcll}
\text{Value} & = & \text{num } n & n \in \mathbb{N} \\
 & & \text{ref } r & r \in \text{Reference} \\
 & & \text{null} & \\
\text{Stack} & = & \text{Value}^* & \\
\text{LocalVar} & = & \text{Var} \to \text{Value} & \\
\text{Frame} & = & \text{ProgCount} \times \text{nameMethod} \times \text{LocalVar} \times \text{Stack} & \\
\text{CallStack} & = & \text{Frame}^* & \\
\text{Object} & = & \text{nameClass} \times (\text{FieldName} \to \text{Value}) & \\
\text{Heap} & = & \text{Reference} \to \text{Object}_\perp & \\
\text{State} & = & \text{Heap} \times \text{CallStack} &
\end{array}
$$

Operational semantics

Basic stack operations

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{nop}}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{const}\ c}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, c :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{pop}}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{dup}}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, v :: v :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{dup2}}{\langle\!\langle h, \langle m, pc, l, v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, v_1 :: v_2 :: v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{swap}}{\langle\!\langle h, \langle m, pc, l, v_1 :: v_2 :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, v_2 :: v_1 :: s\rangle :: sf\rangle\!\rangle}$$

Operational semantics

Arithmetic and local variables.

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{numop}\ op}{\langle\!\langle h, \langle m, pc, l, n_1 :: n_2 :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, [\![op]\!](n_1, n_2) :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{load}\ x}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, l[x] :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{store}\ x}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l[x \mapsto v], s\rangle :: sf\rangle\!\rangle}$$

Operational semantics

Conditionals and control transfer

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{ifle}\ pc' \qquad n \leqslant 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc', l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{ifle}\ pc' \qquad n > 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc + 1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m, pc) = \texttt{goto}\ pc'}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc', l, s\rangle :: sf\rangle\!\rangle}$$

Real Java byte code contains many more "if"s: `if_eq`, `if_ge`, `if_nonnull`, `if_acmpeq`,...

Operational semantics

Objects:

$$\frac{\begin{array}{c}\mathrm{instr}_P(m,pc) = \texttt{new}\ cl \\ h(loc) = \bot \qquad o = (cl, [f \mapsto \diamond]) \qquad h' = h[loc \mapsto o]\end{array}}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h', \langle m, pc+1, l, loc :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m,pc) = \texttt{putfield}\, f \qquad h(loc) = o \qquad o' = o[f \mapsto v]}{\langle\!\langle h, \langle m, pc, l, v :: loc :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h[loc \mapsto o'], \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m,pc) = \texttt{getfield}\, f \qquad h(loc) = o \qquad o = (cl, fld)}{\langle\!\langle h, \langle m, pc, l, loc :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m, pc+1, l, fld(f) :: s\rangle :: sf\rangle\!\rangle}$$

The initial value $\diamond$ of an object field is

- 0 for integers
- `null` for object references.

Operational semantics

Method calls

$$\frac{\begin{array}{c}\mathrm{instr}_P(m,pc) = \texttt{invokevirtual}\ M \\ h(loc) = o \qquad\qquad m' = \mathrm{Lookup}(M, class(o)) \\ f' = \langle m', 1, [r_0 \mapsto loc, r_i \mapsto V_i], \varepsilon\rangle \qquad f'' = \langle m, pc, l, s\rangle\end{array}}{\langle\!\langle h, \langle m, pc, l, loc :: V :: s\rangle :: sf\rangle\!\rangle \to \langle\!\langle h, f' :: f'' :: sf\rangle\!\rangle}$$

$$\frac{\mathrm{instr}_P(m,pc) = \texttt{return} \qquad f' = \langle m', pc', l', s'\rangle}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: f' :: sf\rangle\!\rangle \to \langle\!\langle h, \langle m', pc'+1, l', v :: s'\rangle :: sf\rangle\!\rangle}$$

Other language features

Aspects of the language not dealt with here include:

- Visibility modifiers (`public`, `private`, `protected`,...)
- Multi-threading (`monitorenter`, `monitorexit`)
- Exceptions
- The constant pool
- Arrays
- Long integers

Java byte code verification

Java byte code verification is done at class loading time.

What is verified:

- structural correctness of class file,
- no jumps out of methods,
- size of operand stack at a program point never changes,
- operators get arguments of correct type,
- no pointer arithmetic or forging of references,
- objects and local variables are **initialized** before being used.
- `final` methods are not re-defined.

What is not verified

The Java byte code verifier (BCV) have fewer types that the Java type system

- limited support of booleans,
- no generic types,
- no inner classes,
- interfaces

The BCV leaves a certain number of properties to be checked dynamically:

- null pointer dereferencing,
- array stores and indexing,
- division by zero,
- class casting,
- interface method calls

Typing as a data flow problem

Verification checks that arguments to operands are of correct type.

eg., adding a reference and an integer is not correct

The byte code verifier will

- abstract objects by their class
- not distinguish between different method calls

Compute for each program point

- a type for each local variable (register) and
- a type for the operand stack

This can be seen as a **data flow analysis** where data is replaced by types.

The abstract state at each program point is

$$R : Int \to Types \quad , \quad S : Types^*$$

Example of typing

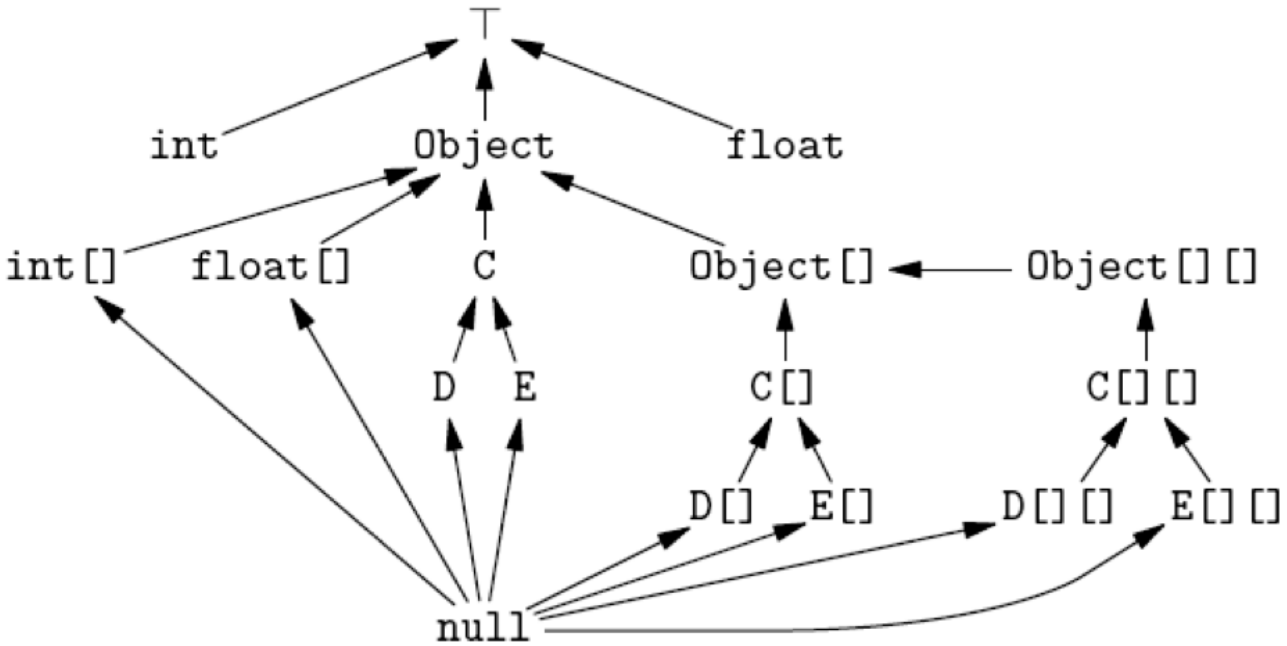
The byte code verifier will compute information like:

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //  R: int,int   S: []
11: goto 2 //  R: int,int   S: []
14: iload 1 //  R: int,int   S: []
15: ireturn //  R: int,int   S: [int]
```

The $\sqcup$-semi lattice of types

The class hierarchy is extended with an element $\top$ ("no type information"), and an element representing `null` values.

The sub-typing relation `<:` is defined by the following Hasse diagram.



The lattice of abstract states

The $\sqcup$ operator is extended component-wise to abstract states of form:

$$R : Int \to Types \quad , \quad S : Types^*$$

Least upper bound of two sets of registers is done pointwise.

Least upper bound of two **stacks** is the pointwise extension of $\sqcup$ on types **provided** the stacks are of same size, otherwise $\top_{State}$.

We add

- a least element $\bot_{State}$ that represents the undefined state.
- a greatest element $\top_{State}$, used to describe an error state.

Typing as data flow analysis

The **initial** state for the verification of a method:

At the first instruction,

- registers correspoding to paramters are given the type of the parameter,
- all other registers are set to type $\top$ (no information),
- and the stack S is empty.

At all other instructions,

- the registers and stacks are undefined ($\bot$)

The abstract state is **propagated** from instruction to instruction following the control flow.

When control flows to an instruction from more than one program point, we take the **least upper bound** of the incoming states.

Typing as data flow analysis

For each byte code instruction we define a transfer function that formalizes how the instruction makes the abstract state evolve.

We specify the functions using a format like:

```
istore n : (R; int:S) → (R[n → int]; S) if 0 <
n < MaxReg
```

Formally, this describes the map from abstract states to abstract states

$$(R;S) \mapsto \begin{cases} (R[n \to \texttt{int}];\ S_1) & \textit{if } S = \textit{int} : S_1 \textit{ and } 0 < n < \texttt{MaxReg} \\ \top_{State} & \textit{otherwise} \end{cases}$$

Transfer functions

Stack and arithmetic instructions

```
iconst n : (R; S)  →  (R; int:S) if |S| < MaxStack
iadd : (R; int:int:S)  →  (R; int:S)
aconst null : (R; S)  →  (R; null:S) if |S|  < MaxStack
```

Control flow instructions

```
ifle j : (R; int:S)  →  (R; S)
goto j : (R; S)  →  (R; S)
```

The validity of the jump label `j` is checked when reading the class file.

Transfer functions

Instructions on registers

```
iload n : (R; S)  →  (R; int:S)
   if 0 < n < MaxReg and R(n) = int and |S|  < MaxStack
istore n : (R; int:S)  →  (R[n  →  int]; S)
   if 0 < n < MaxReg

aload n : (R; S)  →  (R; R(n):S)
   if 0  ⩽  n < MaxReg and R(n) <: Object and |S|  < MaxStack
astore n : (R; t:S)  →  (R[n  →  t]; S)
   if 0 < n < MaxReg and t <: Object
```

Instructions on objects

```
new C : (R;S)  →  (R; C:S)   if |S| < MaxStack

getfield C:f:t : (R; t':S) →  (R; t:S) if t' <: C

putfield C:f:t : (R; t1:t2:S)  →  (R; S)
   if t1 <: t and t2 <: C
```

Recall, <: is the sub-typing relation.

Transfer functions

The method call instruction.

We assume that a method identifier contains its signature `sig` that defines the types of its parameters and its result.

```
invokevirtual C:m:sig : (R; tn : ... : t1 : t0 : S)  →  (R; s:S)
    if sig = s (s1, ... , sn) and
       t0 <: C,  ti <: si for i = 1 ... n
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //
1: istore 1 //
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //  R: int,int   S: []
11: goto 2 //
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //  R: int,int   S: []
11: goto 2 //  R: int,int   S: []
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //    R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //  R: int,int   S: []
11: goto 2 //  R: int,int   S: []
14: iload 1 //  R: int,int   S: []
15: ireturn //
```

Example: typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  R: int,⊤   S: []
1: istore 1 //  R: int,⊤   S: [int]
2: iload 0 //  R: int,int   S: []
3: ifle 14 //  R: int,int   S: [int]
6: iload 1 //  R: int,int   S: []
7: iload 0 //  R: int,int   S: [int]
8: imul //  R: int,int   S: [int :: int]
9: istore 1 //  R: int,int   S: [int]
10: iinc 0, -1 //  R: int,int   S: []
11: goto 2 //  R: int,int   S: []
14: iload 1 //  R: int,int   S: []
15: ireturn //  R: int,int   S: [int]
```

Exercise

Consider the following bytecode program

```
0:  ifle 6
1:  iload 1
2:  iconst 1
3:  iadd
4:  istore 2
5:  goto 9
6:  new Object
7:  astore 2
8:  goto 9
9:  iload 1
```

Show that this program is typable by the bytecode verifier with `MaxStack` $= 3$ and `MaxReg` $= 3$, starting with an operand stack [`int`] (one element of type `int`) and a register 1 of type `int`.

Object creation and initialization

In Java byte code, objects of class `C` are created in two steps:

- they are *allocated* by `new`
- and *initialized* by a call to the constructor method `<init>` of the class `C`.

```
new C   // create uninitialized instance of class C
dup     // duplicate reference
....    // compute args to constructor
invokespecial C.<init>
```

The Java byte code language definition says:

> *It is an* **`error`** *to use an object (except assigning values to local fields) before all its constructors have been called.*

Similarly, it is an error to read from a local variable (a register) before it has been assigned a value (no `load` from a register of type $\top$).

Object initialization analysis

The byte code verifier performs two static analyses to check

1. that a constructor of a class always calls a constructor of the super-class.
2. that an object is not used before one of its constructors has been called,

For the first analysis:

- Add an extra element to the state, that is set to *initialized* when a constructor of the super class is called.

For the second analysis:

- Mark object types as "not yet completely initialized": $\overline{C}$
- Add an instruction number to distinguish between objects allocated at different program points: $\overline{C}_4$.
- Change status of all objects of a given type when exiting the constructor of that method.

Object initialization analysis

Example:

```
 0: new C      // stack type after: C̄0
 3: dup                          //    C̄0, C̄0
 4: new C                        //    C̄0, C̄0, C̄4
 7: dup                          //    C̄0, C̄0, C̄4, C̄4
 8: aconst_null                  //    C̄0, C̄0, C̄4, C̄4, null
 9: invokespecial C.<init>       //    C̄0, C̄0, C
12: invokespecial C.<init>       //    C
15: ...
```

Need restriction to get alias analysis right:

> *no stack element of type* $\overline{C}_p$ *when executing the instruction* `new C` *at program point* p.

Java interfaces

A Java interface specifes methods and fields but not their implementation.

Classes can be declared to implement one or more interfaces.

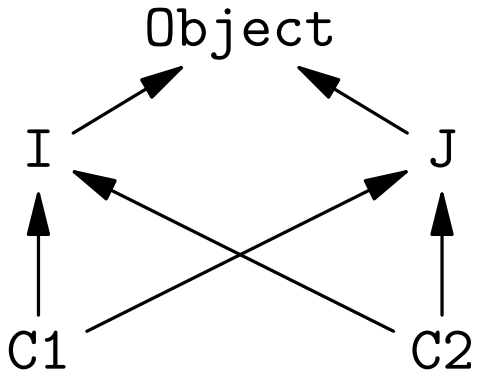
Interfaces can be used as declared types for variable.

```
interface PointInterface {
        void move(int dx, int dy);
}

public class C {
   PointInterface x;
...
   x.move(4,2);
}
```

The problem with interfaces

Classes can implement **several** interfaces

$\Rightarrow$ certain least upper bounds do not exist.

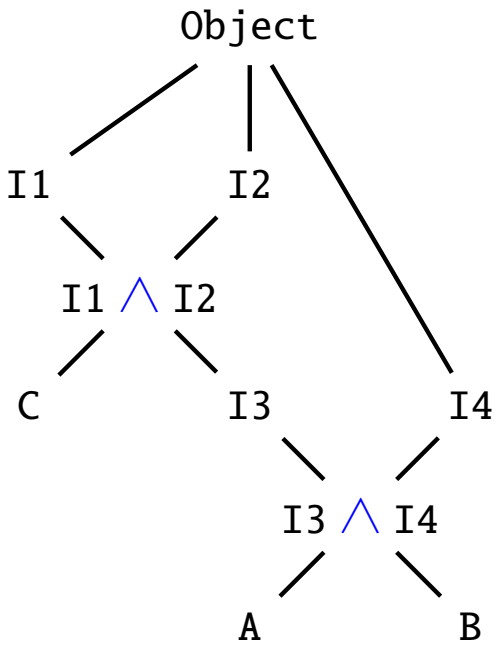


Implemented solution: treat interfaces as `Objects` and check at run-time whether an object implements an interfaces.

Another option: introduce intersection types to join several interfaces into one type.

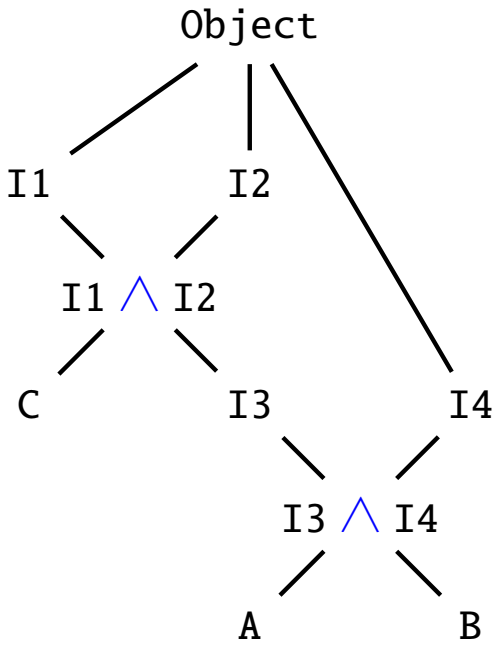
BCV with interfaces

- `C` implements `I1` and `I2`
- `I3` extends `I1` and `I2`
- `A` and `B` implements `I3` and `I4`
- Each `Ii` declares a method `mi`.



BCV with interfaces

- `C` implements `I1` and `I2`
- `I3` extends `I1` and `I2`
- `A` and `B` implements `I3` and `I4`
- Each `Ii` declares a method `mi`.



Exercise: find a type for `i`

```
void foo(boolean b) {
 if (b) {
   i:=new A() ;
 } else
   i:=new B();
...
i.m1();
i.m2();
}
```

BCV with interfaces

Standard BCV will not be able to infer a type for variable `i`.

Inference with intersection types will infer `I1` $\wedge$ `I2`

Notice: there is a weaker intersection-free type for `i`: `I3`

Prune typings to obtain stack maps without intersection types

- work for most (but not all!) byte codes
- Eclipse can be given stack maps w/o intersections

Verification of sub-routines

`jsr l` : jump to program point `l`, pushing the address of the follwing instruction

`ret n` : recover a return address from register `n` and jump to it.

Problems:
- sub-routine entries are *merge points*
- may limit precision

This complicates the byte code verification —
for a relatively **little gain**.

```
0: jsr 100 // register 0 undef
3: ...
50: iconst 0
51: istore 0
52: jsr 100  // register 0 int
55: iload 0
56: ireturn
...
100: astore 1
101: ...  // don't touch register 0
110: ret 1
```

Summary

Java byte code verification

- checks `.class` files when loaded into a Java virtual machine
- part of the security architecture of Java
- verifies
  - well typing
  - object initialization

We have treated a subset of Java BC here

- basic sequential, procedural Java byte code
- interfaces and subroutines
- we did not deal with exceptions and arrays

Literature

Xavier Leroy: Bytecode verification: algorithms and formalizations, J. Automated Reasoning, 30(3–4), 2003.

Stephen N. Freund, John C. Mitchell: A Type System for the Java Bytecode Language and Verifier. J. Automated Reasoning 30(3-4), 2003.

Gerwin Klein, Tobias Nipkow: Verified bytecode verifiers, Theoretical Computer Science, 298(3), 2003.

G. Barthe, G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. Proc. of FASE'04, LNCS 2984, 2004.