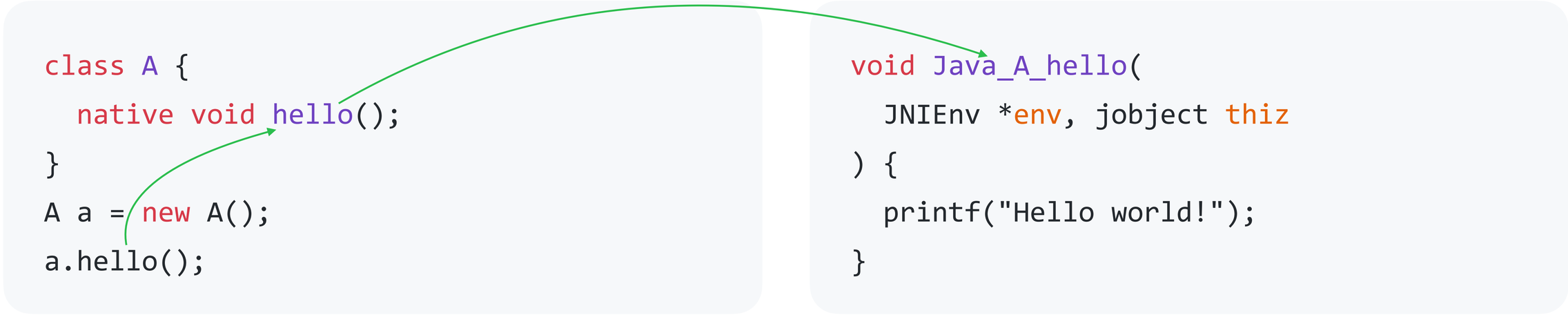
tijni

A static analyzer detecting use-after-free and double-free bugs in JNI programs

Jaemin Hong
jaemin.hong@kaist.ac.kr

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs



```java
class A {
  native void hello();
}
A a = new A();
a.hello();
```

A.java

```c
void Java_A_hello(
  JNIEnv *env, jobject thiz
) {
  printf("Hello world!");
}
```

a.c

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```java
class A {
  long p;

  native long nAllocate();
  native void nUse(... long ptr);
  native void nDeallocate(long ptr);
  void allocate() { p = nAllocate(); }
  void use(...) { nUse(... p); }
  void deallocate() { nDeallocate(p); }
}
```

A.java

```c
struct a;
jlong Java_A_nAllocate(...) {
  return (jlong) malloc(sizeof(struct a));
}
void Java_A_nUse(... jlong ptr) {
   ... (struct a*) ptr ...
}
void Java_A_nDeallocate(... jlong ptr) {
  free((void *) ptr);
}
```

a.c

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```java
class A {
  long p;

  native long nAllocate();
  native void nUse(... long ptr);
  native void nDeallocate(long ptr);
  void allocate() { p = nAllocate(); }
  void use(...) { nUse(... p); }
  void deallocate() { nDeallocate(p); }
}
```

A.java

```c
struct a;
jlong Java_A_nAllocate(...) {
  return (jlong) malloc(sizeof(struct a));
}
void Java_A_nUse(... jlong ptr) {
   ... (struct a*) ptr ...
}
void Java_A_nDeallocate(... jlong ptr) {
  free((void *) ptr);
}
```

a.c

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```java
class A {
  long p;

  native long nAllocate();
  native void nUse(... long ptr);
  native void nDeallocate(long ptr);
  void allocate() { p = nAllocate(); }
  void use(...) { nUse(... p); }
  void deallocate() { nDeallocate(p); }
}
```

A.java

```c
struct a;
jlong Java_A_nAllocate(...) {
  return (jlong) malloc(sizeof(struct a));
}
void Java_A_nUse(... jlong ptr) {
   ... (struct a*) ptr ...
}
void Java_A_nDeallocate(... jlong ptr) {
  free((void *) ptr);
}
```

a.c

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```java
class A {
  long p;

  native long nAllocate();
  native void nUse(... long ptr);
  native void nDeallocate(long ptr);
  void allocate() { p = nAllocate(); }
  void use(...) { nUse(... p); }
  void deallocate() { nDeallocate(p); }
}
```

A.java

```c
struct a;
jlong Java_A_nAllocate(...) {
  return (jlong) malloc(sizeof(struct a));
}
void Java_A_nUse(... jlong ptr) {
   ... (struct a*) ptr ...
}
void Java_A_nDeallocate(... jlong ptr) {
  free((void *) ptr);
}
```

a.c

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```
A a = new A();
a.allocate();

a.use(...);
a.use(...);

a.deallocate();

a.use(...);  // use-after-free
a.deallocate();  // double-free
```

Peer Class Idiom

Making a wrapper class for a C / C++ struct

```
A a = new A();
a.allocate();

a.use(...);
a.use(...);

a.deallocate();

a.use(...);  // use-after-free
a.deallocate();  // double-free
```

```
java(88116,0x70000c587000) malloc: *** error for object
0x7ff5afd309d0: pointer being freed was not allocated
java(88116,0x70000c587000) malloc: *** set a breakpoint
in malloc_error_break to debug
```

The Goal of the Project

Design and implement a **static** analyzer that detects

use-after-free and **double-free** bugs

made by **Java programmers**

in **JNI** programs.

Challenge

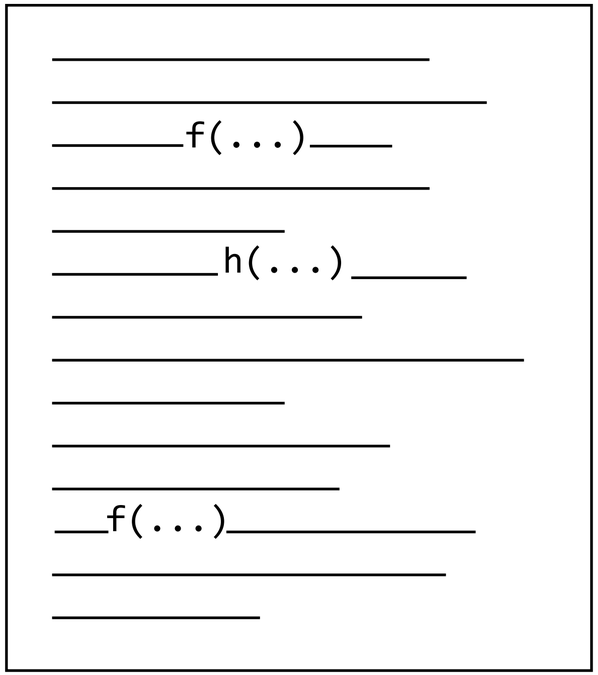
JNI programs are written in more than one language.

Java programs and native programs have different execution contexts.

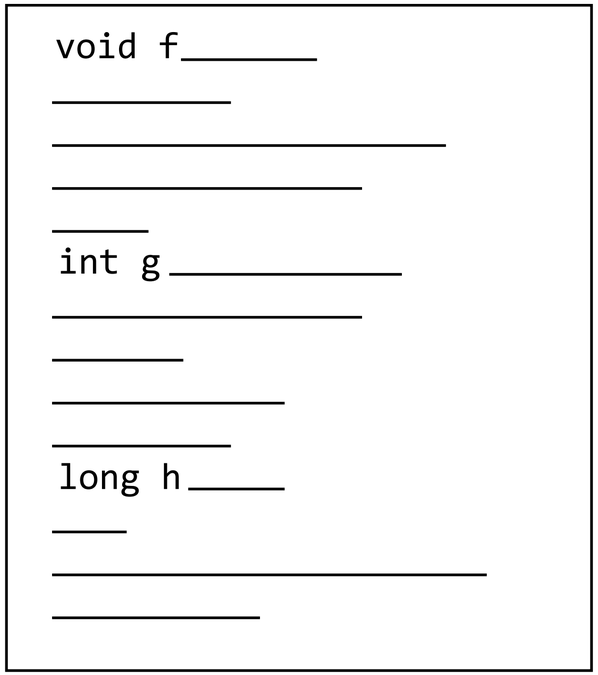
Naively designing a single top-down analyzer for an entire program would not work...

Approach

Making summaries for native functions



Java code



Native code

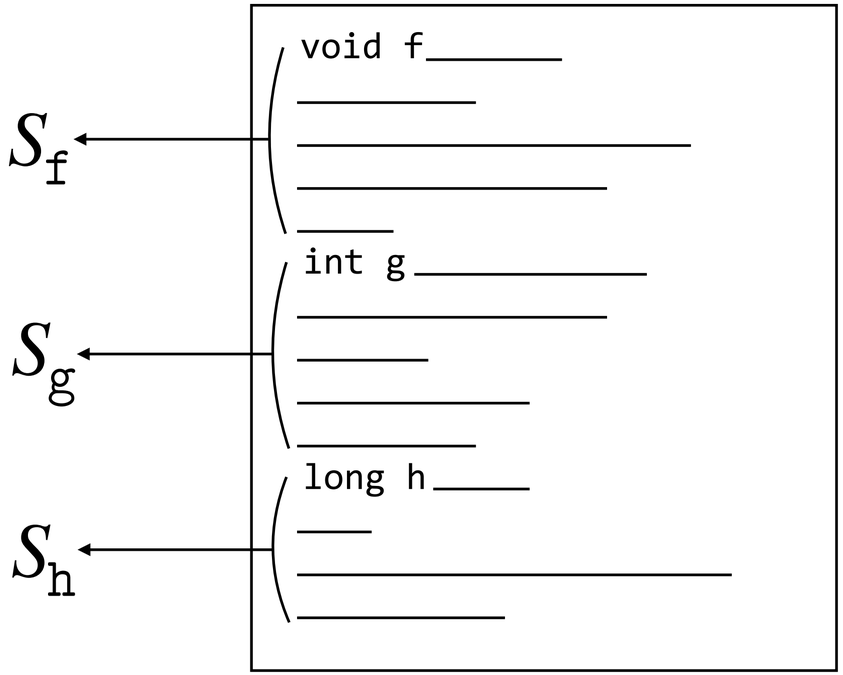
Approach

Making summaries for native functions

Java code

Phase 1

A modular flow-insensitive analysis for native code



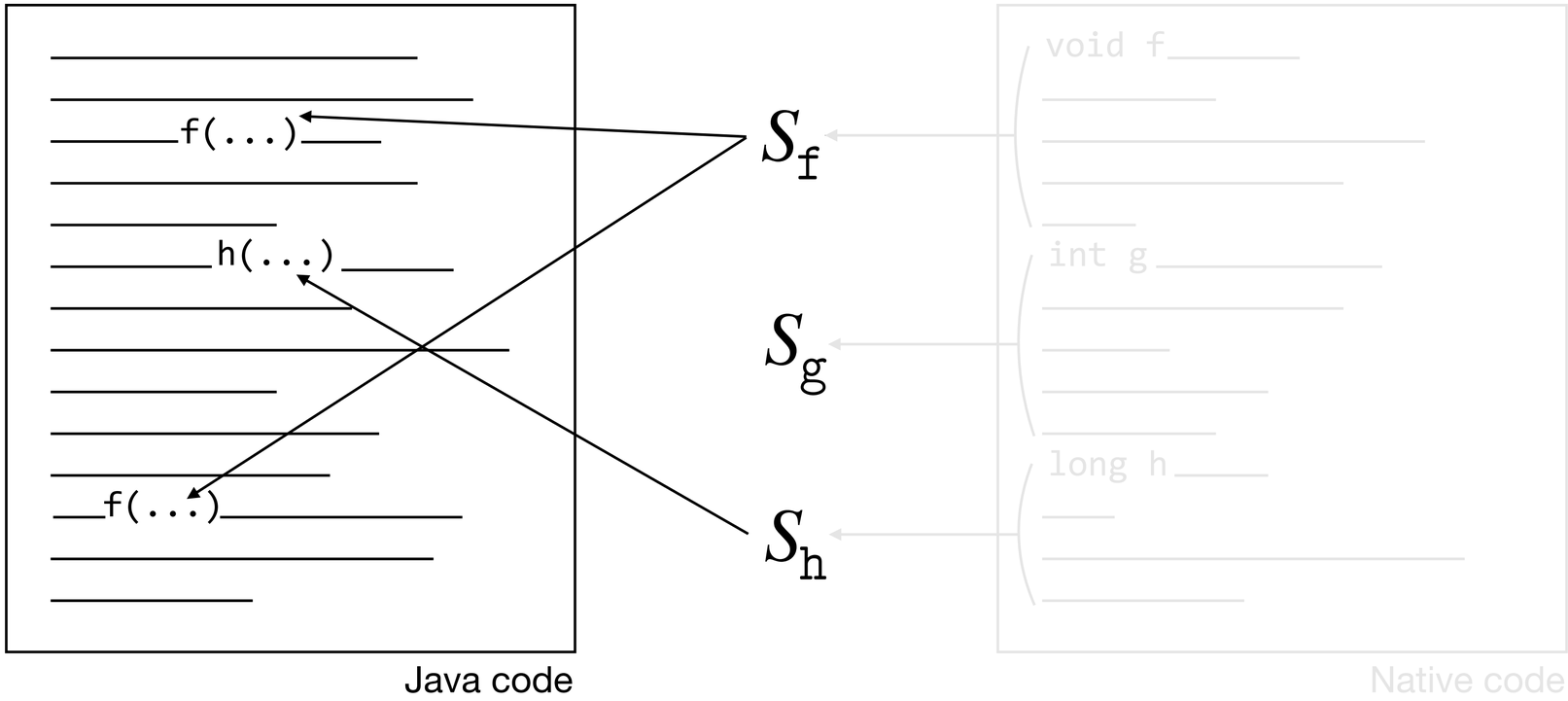
Native code

Approach

Making summaries for native functions

Phase 2

A global analysis for Java code with summaries



Phase 1

A modular flow-insensitive analysis for native code

$$
\begin{array}{rcl}
E & ::= & \cdots \\
 & | & E(E, \ldots, E) \\
 & | & \text{bitcast } E \\
 & | & \text{toptr } E \\
 & | & \text{toint } E \\
C & ::= & \cdots \\
 & | & \text{free } E
\end{array}
$$

Unsoundness Conditions

Recursive functions

Aliased parameters

Global variables

Phase 1

A modular flow-insensitive analysis for native code

l_1:

```
long allocate() {
  return (long) malloc(sizeof(int));
}
```

$$S_{\texttt{allocate}}$$

$$\texttt{returnV} = \{\text{allocsite } l_1\}$$

$$\texttt{used} = \{\}$$

$$\texttt{freed} = \{\}$$

$$\texttt{memory} = []$$

Phase 1

A modular flow-insensitive analysis for native code

```
void deallocate(long ptr) {    [ptr ↦ 'ptr]
  free((void *) ptr);
}
```

$$
\begin{aligned}
S_{\texttt{deallocate}} \\
\texttt{returnV} &= \{\} \\
\texttt{used} &= \{\} \\
\texttt{freed} &= \{\texttt{'ptr}\} \\
\texttt{memory} &= [\texttt{ptr} \mapsto \texttt{'ptr}]
\end{aligned}
$$

Phase 1

A modular flow-insensitive analysis for native code

```
long read(long ptr) {  [ptr ↦ ′ptr, ′ptr ↦ ″ptr]
  return *((long *) ptr);
}
```

$$
\begin{aligned}
& S_{\texttt{read}} \\
\texttt{returnV} &= \{''\texttt{ptr}\} \\
\texttt{used} &= \{'\texttt{ptr}\} \\
\texttt{freed} &= \{\} \\
\texttt{memory} &= [\texttt{ptr} \mapsto '\texttt{ptr}, '\texttt{ptr} \mapsto ''\texttt{ptr}]
\end{aligned}
$$

Java Bytecode Overview

$$
\begin{array}{lcll}
T & \in & \mathbb{T} & \text{class} \\
n & \in & \mathbb{Z} & \\
i & \in & \mathbb{N} & \\
f & \in & \mathbb{F} & \text{field} \\
w & \in & \mathbb{W} & \text{method} \\
l & \in & \mathbb{L} & \text{label}
\end{array}
\qquad
\begin{array}{lcl}
I & ::= & \text{new } T \mid \text{const } n \\
 & \mid & \text{load } i \mid \text{store } i \\
 & \mid & \text{getfield } f \mid \text{putfield } f \\
 & \mid & \text{invokevirtual } w \\
 & \mid & \text{cmp} \mid \text{ifeq } l \\
 & \mid & \text{goto } l \mid \text{return}
\end{array}
$$

Phase 2

A global analysis for Java code with summaries

$$
\begin{aligned}
c \in \mathbb{C} &= \mathbb{M} \times \mathbb{H} \times \mathbb{G} \times ((\mathbb{W} \times \mathbb{J}) \to \mathbb{K}) \\
\\
m \in \mathbb{M} &= (\mathbb{W} \times \mathbb{J} \times \mathbb{N}) \to \mathbb{V} \\
h \in \mathbb{H} &= \mathbb{A} \to \mathbb{O} \\
g \in \mathbb{G} &= \mathbb{U} \to \mathbb{V} \\
k \in \mathbb{K} & \\
\\
o \in \mathbb{O} &= \mathbb{F} \to \mathbb{V} \\
v \in \mathbb{V} &= \mathbb{Z} \times \mathbb{A} \times \mathbb{U} \\
a \in \mathbb{A} & \quad \text{heap adderss} \\
u \in \mathbb{U} & \quad \text{C heap adderss} \\
j \in \mathbb{J} & \quad \text{instance}
\end{aligned}
$$

$$
\begin{aligned}
\langle l, m^{\#} \rangle \in \mathbb{S}^{\#} &= \mathbb{L} \times \mathbb{C}^{\#} \\
c^{\#} \in \mathbb{C}^{\#} &= \mathbb{M}^{\#} \times \mathbb{H}^{\#} \times \mathbb{G}^{\#} \times \mathbb{U}^{\#} \times (\mathbb{W} \to \mathbb{K}^{\#}) \\
\\
m^{\#} \in \mathbb{M}^{\#} &= (\mathbb{W} \times \mathbb{N}) \to \mathbb{V}^{\#} \\
h^{\#} \in \mathbb{H}^{\#} &= \mathbb{A}^{\#} \to \mathbb{O}^{\#} \\
g^{\#} \in \mathbb{G}^{\#} &= \mathbb{U}^{\#} \to \mathbb{V}^{\#} \\
k^{\#} \in \mathbb{K}^{\#} & \\
\\
o^{\#} \in \mathbb{O}^{\#} &= \mathbb{F} \to \mathbb{V}^{\#} \\
v^{\#} \in \mathbb{V}^{\#} &= \mathbb{Z}^{\#} \times \mathbb{A}^{\#} \times \mathbb{U}^{\#}
\end{aligned}
$$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}

l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l4:   return (jlong) malloc(sizeof(long));
  }
  void Java_Box_deallocate(... jlong p) {
l5:   free((void *) p);
  }
```

Heap	C Heap	Freed
[]	[]	⊥

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}

l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l4:   return (jlong) malloc(sizeof(long));
  }
  void Java_Box_deallocate(... jlong p) {
l5:   free((void *) p);
  }
```

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[]	$\bot$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}

l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.deallocate(p);
```

```
  jlong
l4:   retu                                    );
  }
  void Java_Box_deallocate(... jlong p) {
l5:   free((void *) p);
  }
```

$$S_{\texttt{allocate}}$$
$$\texttt{returnV} = \{\text{allocsite } l_4\}$$

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\rangle$]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\rangle$]	[allocsite $l_4 \mapsto \{'\text{allocsite } l_4\}$]	$\bot$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}

l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l4:   return (jlong) malloc(sizeof(long));
  }
  void J                                  ) {
l5:   free
  }
```

$S_{\texttt{deallocate}}$
$\texttt{freed} = \{'\texttt{ptr}\}$

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\rangle$]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\rangle$]	[allocsite $l_4 \mapsto \{'\text{allocsite } l_4\}$]	$\bot$
[allocsite $l_1 \mapsto \langle\rangle$]	[allocsite $l_4 \mapsto \{'\text{allocsite } l_4\}$]	{allocsite l_4}

Results

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
Box b = new Box();
long p = b.allocate();
b.write(p, 1);
b.deallocate(p);
```

No alarms!

```
jlong Java_Box_allocate(...) {
  return (jlong) malloc(sizeof(long));
}
void Java_Box_write(... jlong p, jlong v) {
   *((long*) p) = v;
}
void Java_Box_deallocate(... jlong p) {
  free((void *) p);
}
```

Results

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
Box b = new Box();
long p = b.allocate();
b.deallocate(p);
b.write(p, 1);  // use-after-free
```

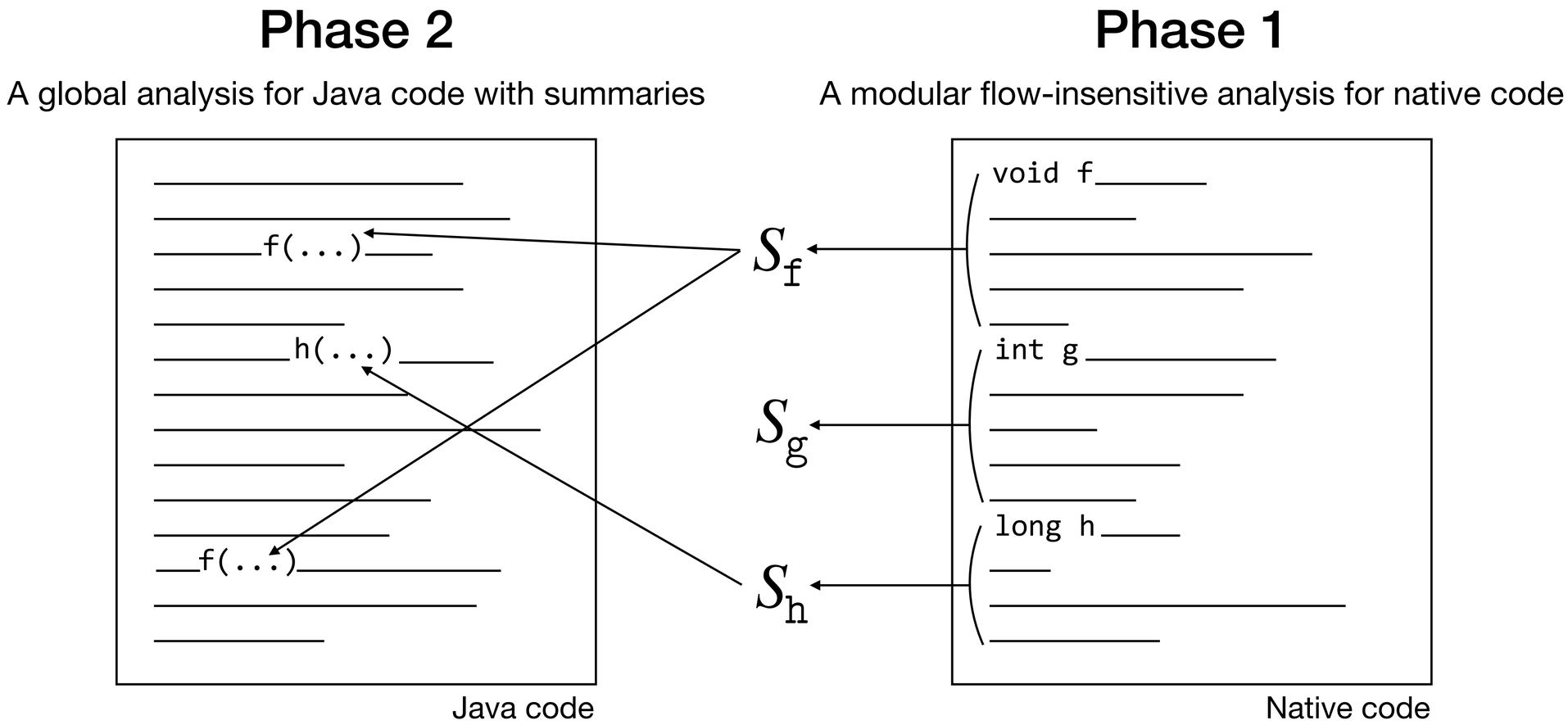
```
jlong Java_Box_allocate(...) {
  return (jlong) malloc(sizeof(long));
}
void Java_Box_write(... jlong p, jlong v) {
   *((long*) p) = v;
}
void Java_Box_deallocate(... jlong p) {
  free((void *) p);
}
```

Results

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
Box b = new Box();
long p = b.allocate();
b.deallocate(p);
b.deallocate(p);  // double-free
```

```
jlong Java_Box_allocate(...) {
  return (jlong) malloc(sizeof(long));
}
void Java_Box_write(... jlong p, jlong v) {
   *((long*) p) = v;
}
void Java_Box_deallocate(... jlong p) {
  free((void *) p);
}
```

tijni: A static analyzer detecting use-after-free and double-free bugs in JNI programs



Jaemin Hong
jaemin.hong@kaist.ac.kr

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs

```java
class A {
  int foo() {
    return 1;
  }
  native int bar(int i);
}

A a = new A();
a.bar(2);
```

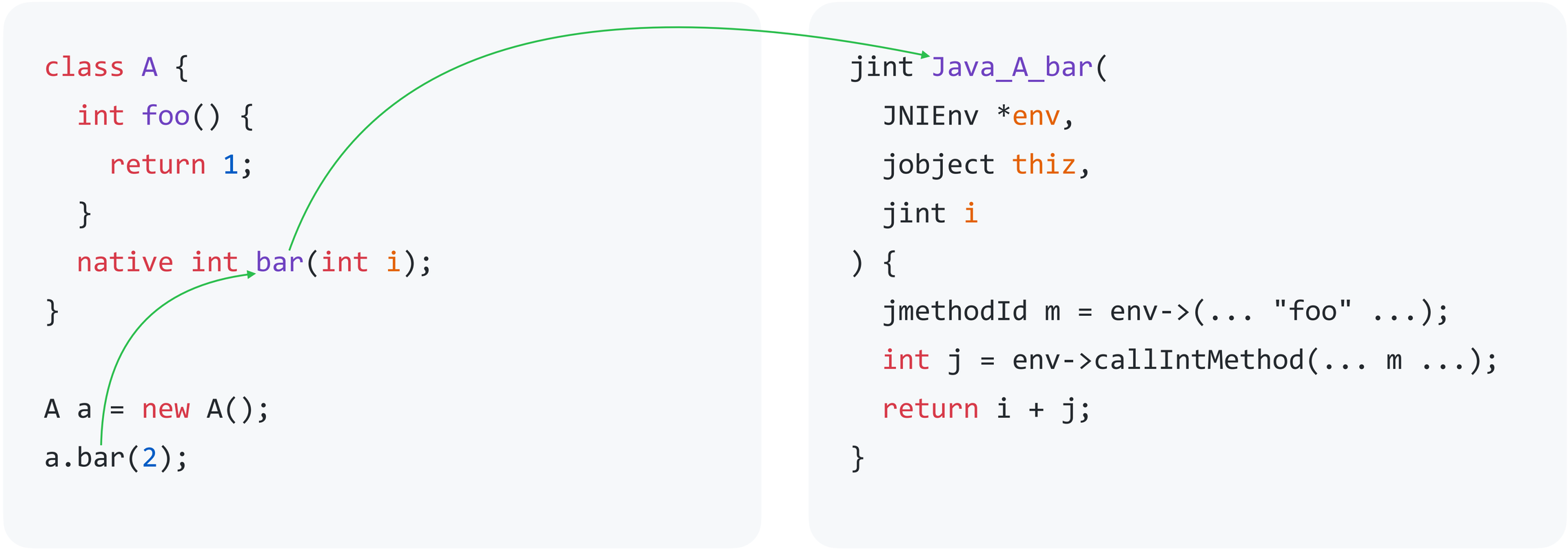
A.java

```c
jint Java_A_bar(
  JNIEnv *env,
  jobject thiz,
  jint i
) {
  jmethodId m = env->(... "foo" ...);
  int j = env->callIntMethod(... m ...);
  return i + j;
}
```

a.c

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs



```java
class A {
  int foo() {
    return 1;
  }
  native int bar(int i);
}

A a = new A();
a.bar(2);
```

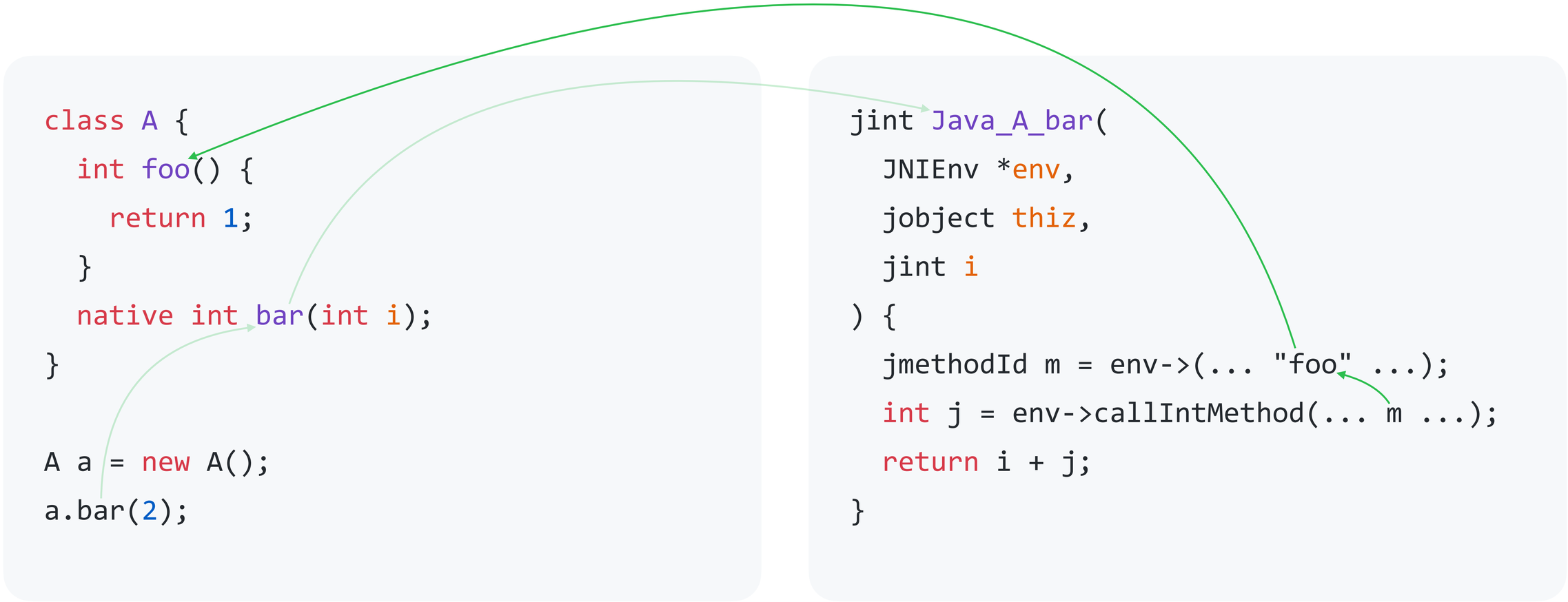
A.java

```c
jint Java_A_bar(
  JNIEnv *env,
  jobject thiz,
  jint i
) {
  jmethodId m = env->(... "foo" ...);
  int j = env->callIntMethod(... m ...);
  return i + j;
}
```

a.c

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs



```java
class A {
  int foo() {
    return 1;
  }
  native int bar(int i);
}

A a = new A();
a.bar(2);
```

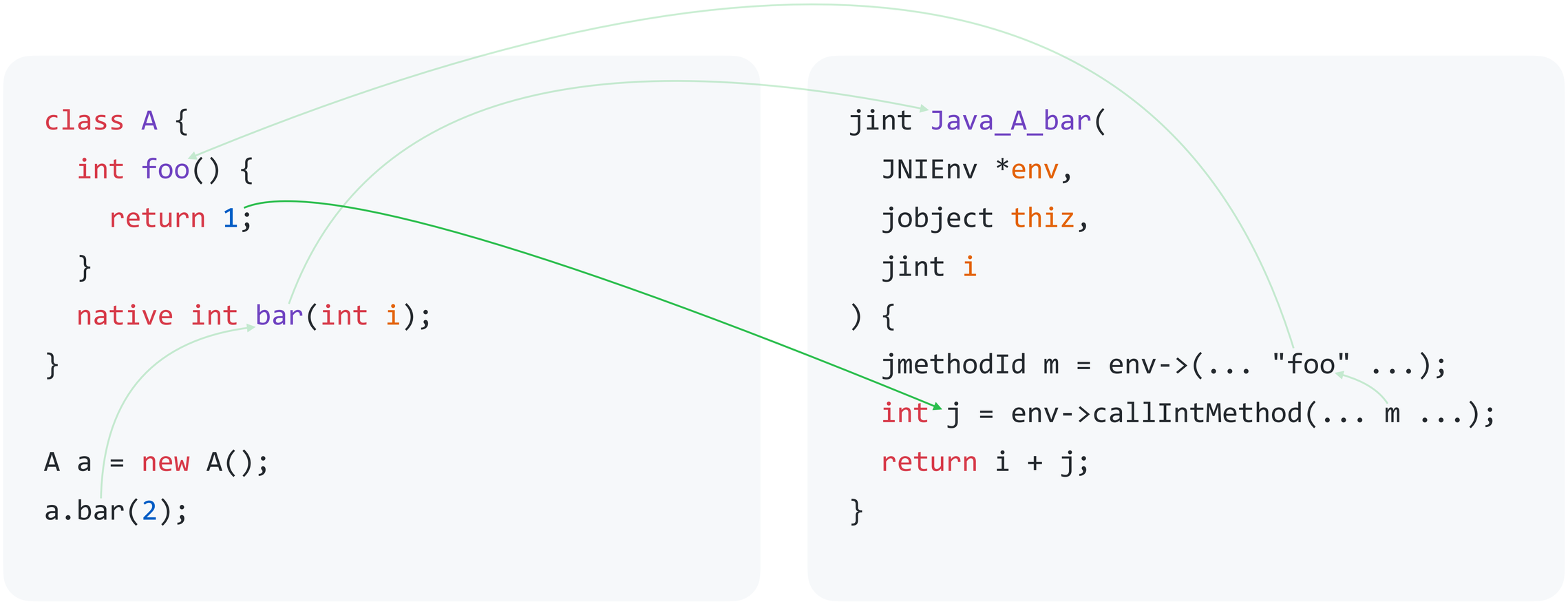
A.java

```c
jint Java_A_bar(
  JNIEnv *env,
  jobject thiz,
  jint i
) {
  jmethodId m = env->(... "foo" ...);
  int j = env->callIntMethod(... m ...);
  return i + j;
}
```

a.c

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs



```java
class A {
  int foo() {
    return 1;
  }
  native int bar(int i);
}

A a = new A();
a.bar(2);
```

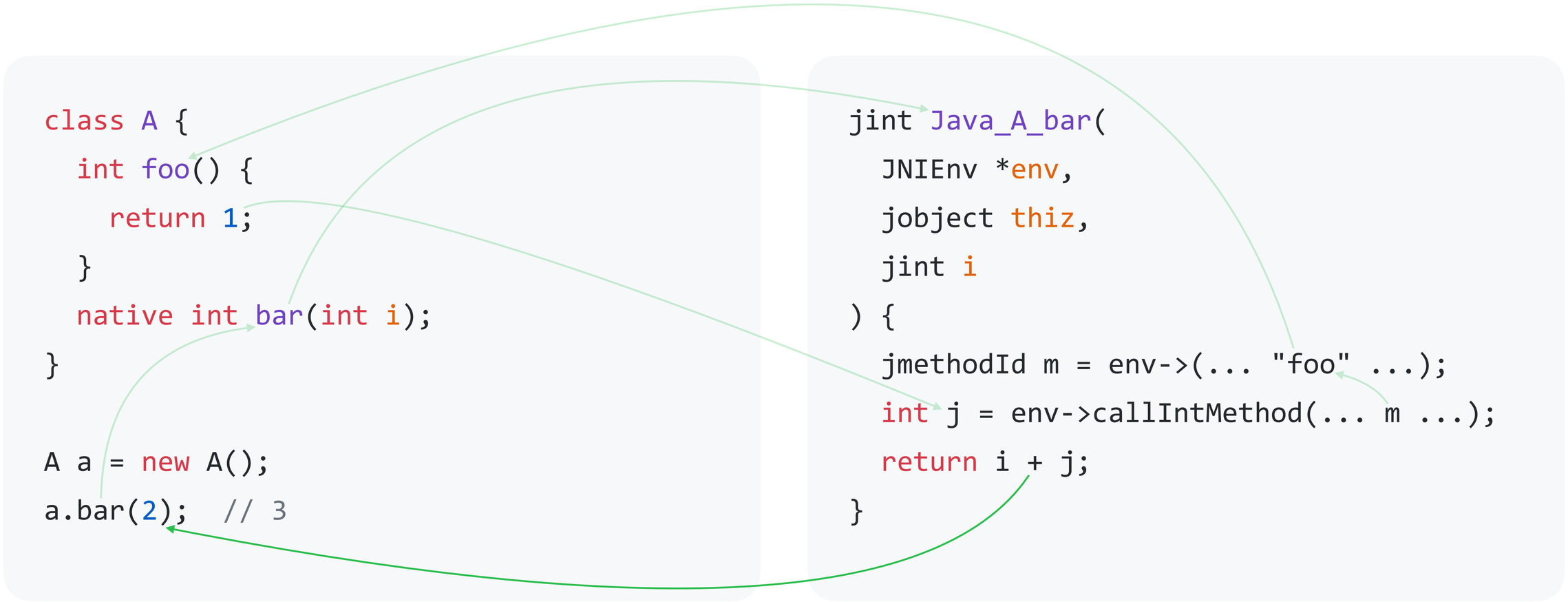
A.java

```c
jint Java_A_bar(
  JNIEnv *env,
  jobject thiz,
  jint i
) {
  jmethodId m = env->(... "foo" ...);
  int j = env->callIntMethod(... m ...);
  return i + j;
}
```

a.c

Java Native Interface (JNI)

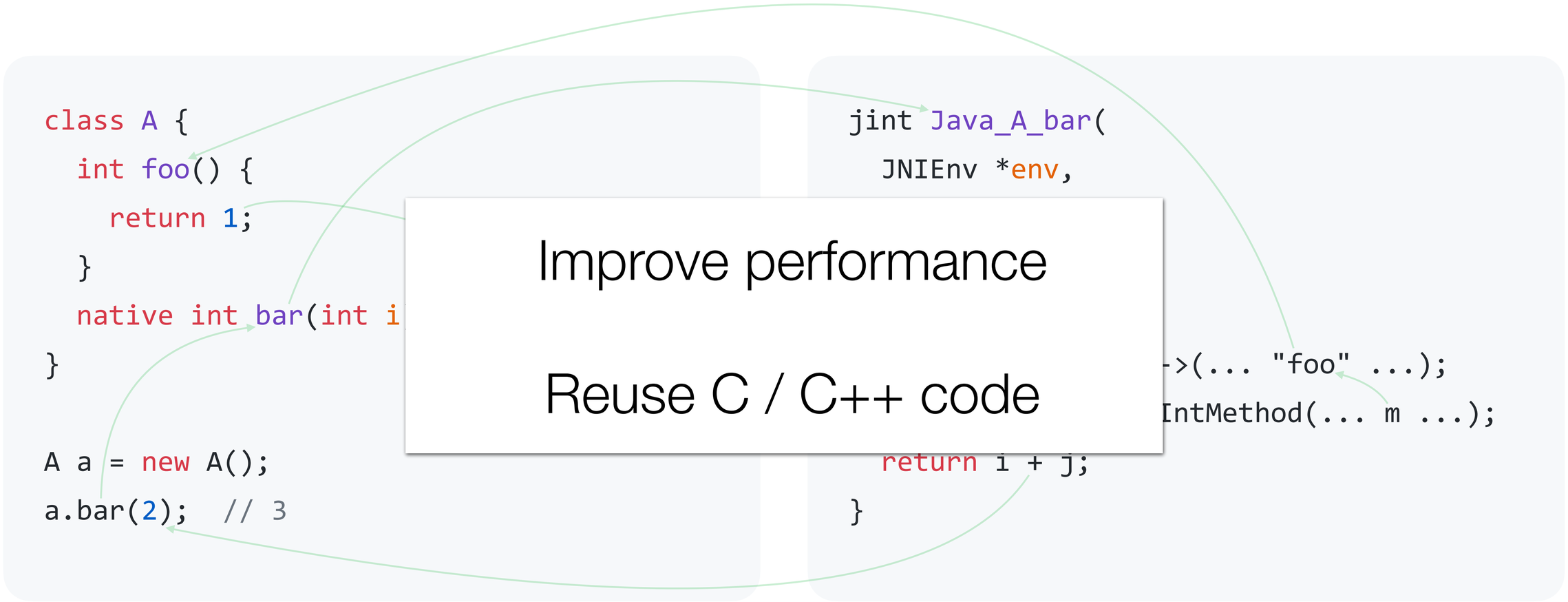
An interface to call C / C++ functions from Java programs



x

Java Native Interface (JNI)

An interface to call C / C++ functions from Java programs



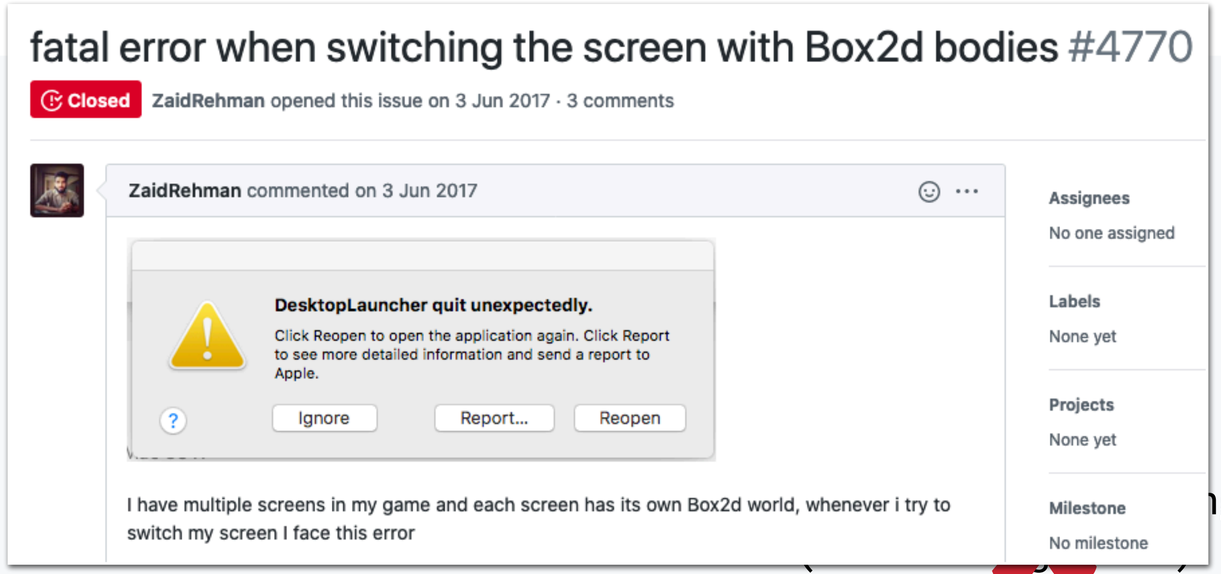
Peer Class Idiom

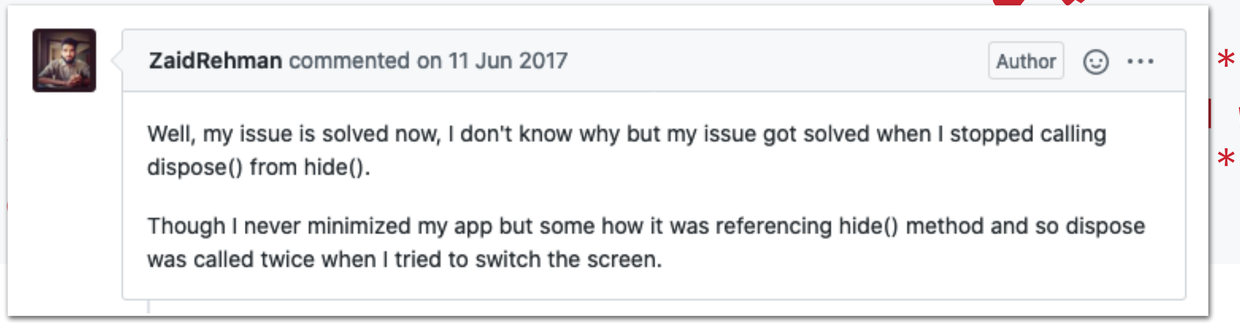
Making a wrapper class for a C / C++ struct

```
A a = new A();
a.allocate();

a.use(...);
a.use(...);

a.deallocate();
...
a.use(...);  // use-
a.deallocate();  //
```






```
ng.NullPointerException

*** error for object
 was not allocated
*** set a breakpoint
```

Phase 1

A modular flow-insensitive analysis for native code

$$[\texttt{ptr} \mapsto {'}\texttt{ptr}, {'}\texttt{ptr} \mapsto {''}\texttt{ptr}, \texttt{value} \mapsto {'}\texttt{value}]$$

```
void write(long ptr, long value) {
  *((long *) ptr) = value;
}
```

$$S_{\texttt{write}}$$

$$\begin{aligned}
\texttt{returnV} &= \{\} \\
\texttt{used} &= \{{'}\texttt{ptr}\} \\
\texttt{freed} &= \{\} \\
\texttt{memory} &= [\texttt{ptr} \mapsto {'}\texttt{ptr}, \\
\texttt{value} &\mapsto {'}\texttt{value}, {'}\texttt{ptr} \mapsto \{{''}\texttt{ptr}, {'}\texttt{value}\}]
\end{aligned}$$

Phase 1

A modular flow-insensitive analysis for native code

$$S_{\texttt{deallocate}}$$
$$\begin{aligned}\texttt{returnV} &= \{\}\\ \texttt{used} &= \{\}\\ \texttt{freed} &= \{'\texttt{ptr}\}\\ \texttt{memory} &= [\texttt{ptr} \mapsto '\texttt{ptr}]\end{aligned}$$

$$S_{\texttt{read}}$$
$$\begin{aligned}\texttt{returnV} &= \{''\texttt{ptr}\}\\ \texttt{used} &= \{'\texttt{ptr}\}\\ \texttt{freed} &= \{\}\\ \texttt{memory} &= [\texttt{ptr} \mapsto '\texttt{ptr}, '\texttt{ptr} \mapsto ''\texttt{ptr}]\end{aligned}$$

```
long read_deallocate(long ptr) {
  long value = read(ptr);
  deallocate(ptr);
  return value;
}
```

$$S_{\texttt{read\_deallocate}}$$
$$\begin{aligned}\texttt{returnV} &= \{''\texttt{ptr}\}\\ \texttt{used} &= \{'\texttt{ptr}\}\\ \texttt{freed} &= \{'\texttt{ptr}\}\\ \texttt{memory} &= [\texttt{ptr} \mapsto '\texttt{ptr}, '\texttt{ptr} \mapsto ''\texttt{ptr}]\end{aligned}$$

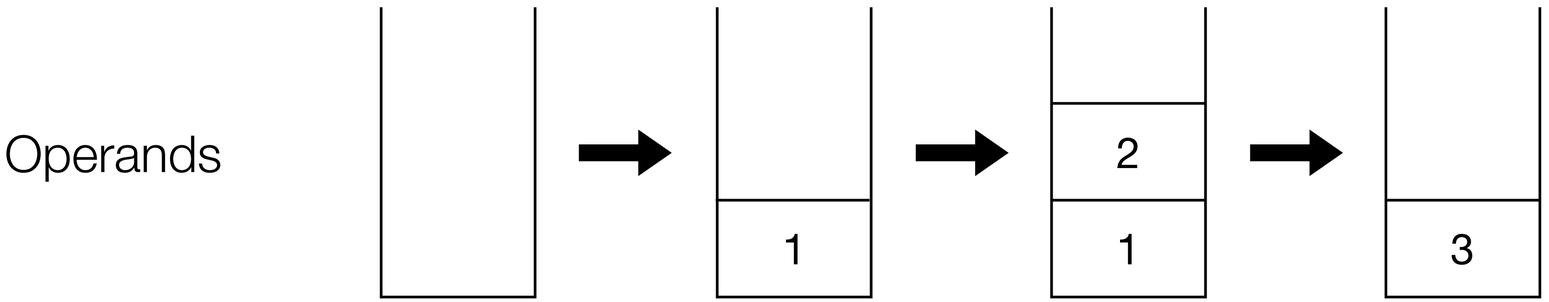
Java Bytecode Overview

```
1 + 2
```

A.java

```
const 1
const 2
add
```

A.class



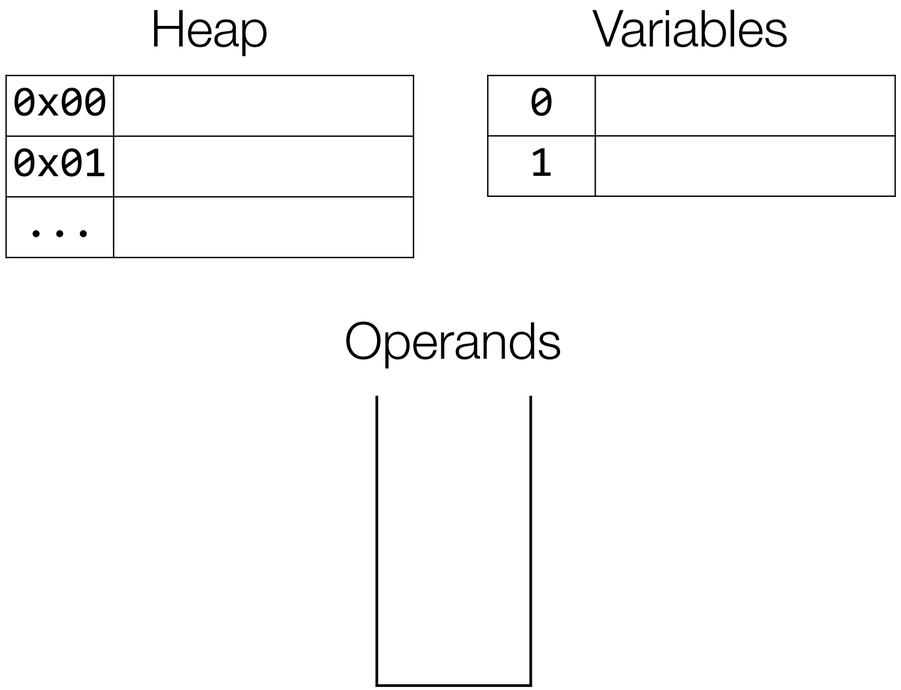
Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

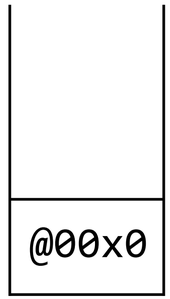
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	
1	

Operands



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

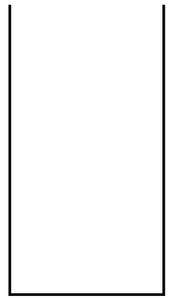
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	@0x00
1	

Operands



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

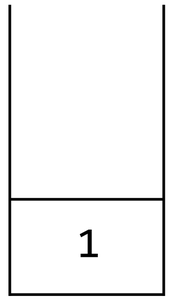
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	@0x00
1	

Operands



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

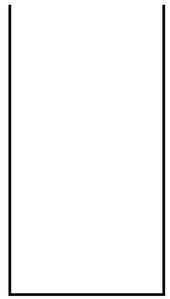
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	@0x00
1	1

Operands



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

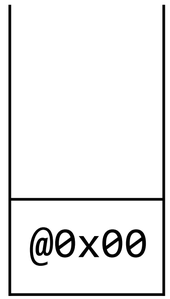
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	@0x00
1	1

Operands



Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```

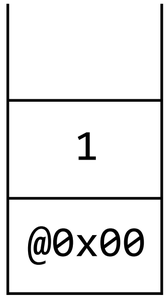
Heap

0x00	{ x = 0 }
0x01	
...	

Variables

0	@0x00
1	1

Operands



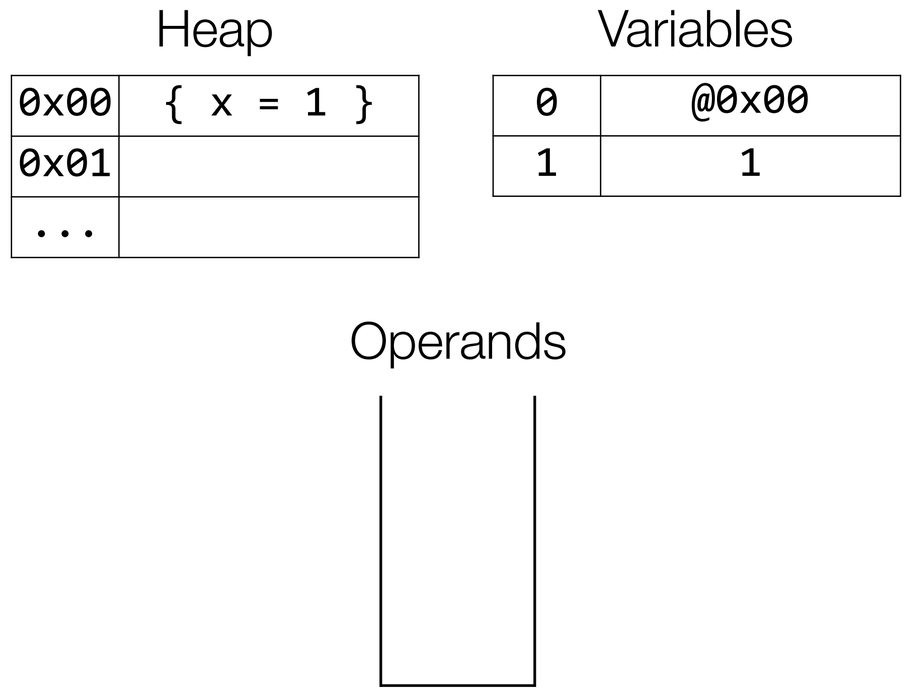
Java Bytecode Overview

```
class A { long x; }
A a = new A();

long l = 1;

a.x = l;
```

```
new A
store 0
const 1
store 1
load 0
load 1
putfield x
```



Phase 2

A global analysis for Java code with summaries

- Abstracting operand stacks

$$
\begin{array}{rcl}
k & \in & \mathbb{K} \\
k & ::= & v; k \mid \cdot
\end{array}
$$

$$
\begin{array}{rcl}
push(k, v) & = & v; k \\
top(v; k) & = & v \\
pop(v; k) & = & k
\end{array}
$$

Phase 2

A global analysis for Java code with summaries

- Abstracting operand stacks

$$
\begin{array}{rcl}
k & \in & \mathbb{K} \\
k & ::= & v; k \mid \cdot
\end{array}
$$

$$
\begin{array}{rcl}
push(k, v) & = & v; k \\
top(v; k) & = & v \\
pop(v; k) & = & k
\end{array}
$$

$$
\begin{array}{rcl}
k^{\#} & \in & \mathbb{K}^{\#} \\
k^{\#} & ::= & \kappa^{\#} \\
\kappa^{\#} & ::= & v^{\#}; \kappa^{\#} \mid \cdot
\end{array}
$$

$$
\begin{array}{rcl}
push^{\#}(\kappa^{\#}, v^{\#}) & = & v^{\#}; \kappa^{\#} \\
top(v^{\#}; \kappa^{\#}) & = & v^{\#} \\
pop(v^{\#}; \kappa^{\#}) & = & \kappa^{\#}
\end{array}
$$

Phase 2

A global analysis for Java code with summaries

- Abstracting operand stacks

$$
\begin{array}{rcl}
k & \in & \mathbb{K} \\
k & ::= & v; k \mid \cdot
\end{array}
$$

$$
\begin{array}{rcl}
push(k, v) & = & v; k \\
top(v; k) & = & v \\
pop(v; k) & = & k
\end{array}
$$

$$
\begin{array}{rcl}
k^{\#} & \in & \mathbb{K}^{\#} \\
k^{\#} & ::= & \kappa^{\#} \mid [v^{\#}] \\
\kappa^{\#} & ::= & v^{\#}; \kappa^{\#} \mid \cdot
\end{array}
$$

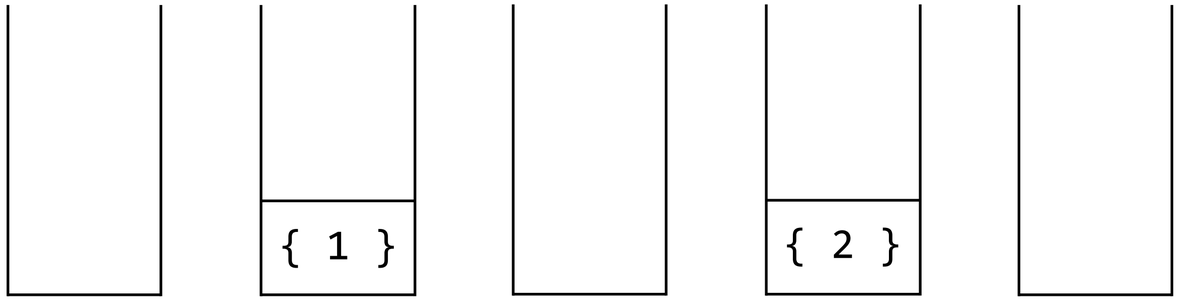
$$
\begin{array}{rcl}
push^{\#}(\kappa^{\#}, v^{\#}) & = & v^{\#}; \kappa^{\#} \\
top(v^{\#}; \kappa^{\#}) & = & v^{\#} \\
pop(v^{\#}; \kappa^{\#}) & = & \kappa^{\#}
\end{array}
$$

$$
\begin{array}{rcl}
push^{\#}([v_1^{\#}], v_2^{\#}) & = & [v_1^{\#} \sqcup v_2^{\#}] \\
top([v^{\#}]) & = & v^{\#} \\
pop([v^{\#}]) & = & [v^{\#}]
\end{array}
$$

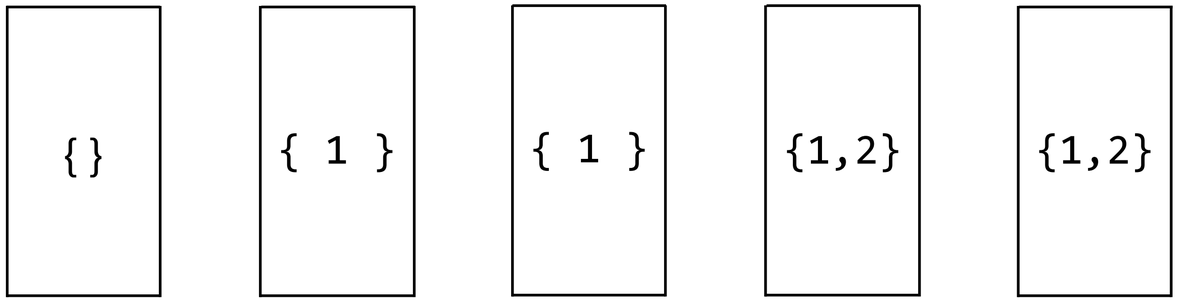
Phase 2

A global analysis for Java code with summaries

```
void f() {
  long x = 1;
  long y = 2;
}
```




```
void f() {
  long x = 1;
  long y = 2;
  f();
}
```



Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
```

l_1: `Box b = new Box();`
l_2: `long p = b.allocate();`
l_3: `b.write(p, 1);`
l_4: `b.deallocate(p);`

```
  jlong Java_Box_allocate(...) {
l5:  return (jlong) malloc(sizeof(long));
  }
  void Java_Box_write(... jlong p, jlong v) {
l6:   *((long*) p) = v;
  }
  void Java_Box_deallocate(... jlong p) {
l7:  free((void *) p);
  }
```

Heap	C Heap	Freed
[]	[]	⊥

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
```

```
l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.write(p, 1);
l4: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l5:   return (jlong) malloc(sizeof(long));
  }
  void Java_Box_write(... jlong p, jlong v) {
l6:    *((long*) p) = v;
  }
  void Java_Box_deallocate(... jlong p) {
l7:   free((void *) p);
  }
```

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[]	$\bot$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
```

```
l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.write(p, 1);
l4: b.deallocate(p);
```

```
  jlong J
l5:   retur
  }
  void Java_Box_write(... jlong p, jlong v) {
l6:    *((long*) p) = v;
  }
  void Java_Box_deallocate(... jlong p) {
l7:   free((void *) p);
  }
```

$$S_{\texttt{allocate}}$$
$$\texttt{returnV} = \{\text{allocsite } l_5\}$$

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{'\text{allocsite } l_5\}$]	$\bot$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
```

```
l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.write(p, 1);
l4: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l5:   return (jlong) malloc(sizeof(long));
  }
  void Java_Box_write(..., jlong p, jlong v) {
l6:   *(long *) p = v;
  }
  void Java_Box_deallocate(..., jlong p) {
l7:   free((void *) p);
  }
```

$$S_{\texttt{write}}$$

$$\texttt{memory} = [\texttt{ptr} \mapsto {'}\texttt{ptr}, \texttt{value} \mapsto {'}\texttt{value}, {'}\texttt{ptr} \mapsto \{{'}\texttt{ptr}, {'}\texttt{value}\}]$$

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{{'}\text{allocsite } l_5\}$]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{{'}\text{allocsite } l_5, 1\}$]	$\bot$

Phase 2

A global analysis for Java code with summaries

```
class Box {
  native long allocate();
  native void write(long p, long v);
  native void deallocate(long p);
}
```

```
l1: Box b = new Box();
l2: long p = b.allocate();
l3: b.write(p, 1);
l4: b.deallocate(p);
```

```
  jlong Java_Box_allocate(...) {
l5:   return (jlong) malloc(sizeof(long));
  }
  void Java_Box_write(... jlong p, jlong v) {
l6:    *((long*) p) = v;
  }
  void Ja
l7:   free(
  }
```

$S_{\texttt{deallocate}}$
$\texttt{freed} = \{'\texttt{ptr}\}$

Heap	C Heap	Freed
[]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{'\text{allocsite } l_5\}$]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{'\text{allocsite } l_5, 1\}$]	$\bot$
[allocsite $l_1 \mapsto \langle\,\rangle$]	[allocsite $l_5 \mapsto \{'\text{allocsite } l_5, 1\}$]	{allocsite l_5}

Results

```
class Box {
  long p;
  native long nAllocate(); void allocate() { p = nAllocate(); }
  native void nWrite(long p, long v); void write(long v) { nWrite(p, v); }
  native void nAllocate(long p); void deallocate() { nDeallocate(p, v);   // double-free }
}
Box b = new Box();
b.allocate();
b.deallocate();
b.deallocate();
```

Context-sensitive analyses can improve results.