Precision-Guided Context Sensitivity for Pointer Analysis

Yue Li, Tian Tan, Anders Møller, Yannis Smaragdakis

OOPSLA 2018

AARHUS UNIVERSITET

National and Kapodistrian University of Athens

A New
Pointer Analysis Technique
for
Object-Oriented Programs

Pointer Analysis

Determines

“which objects a variable can point to?”

Uses of Pointer Analysis

Clients

- Security analysis
- Bug detection
- Compiler optimization
- Program verification
- Program understanding
- …

Tools

Soot

DroidSafe

Doop

Chord

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

…

Uses of Pointer Analysis

Clients

- Security analysis
- Bug detection
- Compiler optimization
- Program verification
- Program understanding
- …

Tools

Soot
DroidSafe
Doop
Chord
WALA
T. J. WATSON LIBRARIES FOR ANALYSIS
…

A precise pointer analysis benefits all above clients & tools

Context Sensitivity

One of the most successful pointer analysis techniques for producing high precision for OO programs

Context Sensitivity

Distinguishes points-to information of methods by different calling contexts

Context Sensitivity: Example

```
class A {
  String foo(String s) {
    return s;
  }
}
```

```
static void main() {
  A a1 = new A(); // A/1
  b1 = a1.foo("s1");

  A a2 = new A(); // A/2
  b2 = a2.foo("s2");
}
```

Variable	Object
s	"s1","s2"
b1	"s1","s2"
b2	"s1","s2"

Context-Insensitivity

Context Sensitivity: Example

```
class A {
  String foo(String s) {
    return s;
  }
}
```

```
static void main() {
  A a1 = new A(); // A/1
  b1 = a1.foo("s1");

  A a2 = new A(); // A/2
  b2 = a2.foo("s2");
}
```

Variable	Object
s	"s1","s2"
b1	"s1","s2"
b2	"s1","s2"

Context-Insensitivity

Context Sensitivity: Example

```
class A {
  String foo(String s) {
    return s;
  }
}
```

```
static void main() {
  A a1 = new A(); // A/1
  b1 = a1.foo("s1");

  A a2 = new A(); // A/2
  b2 = a2.foo("s2");
}
```

Context	Variable	Object
[A/1]	s	"s1"
[A/2]	s	"s2"
[]	b1	"s1"
[]	b2	"s2"

1-Object-Sensitivity

Variable	Object
s	"s1","s2"
b1	"s1","s2"
b2	"s1","s2"

Context-Insensitivity

Context Sensitivity

Widely adopted by static analysis frameworks for OO programs

Doop

Soot

Chord

TAJS

DroidSafe

FlowDroid

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

Problem of Context Sensitivity (C.S.)

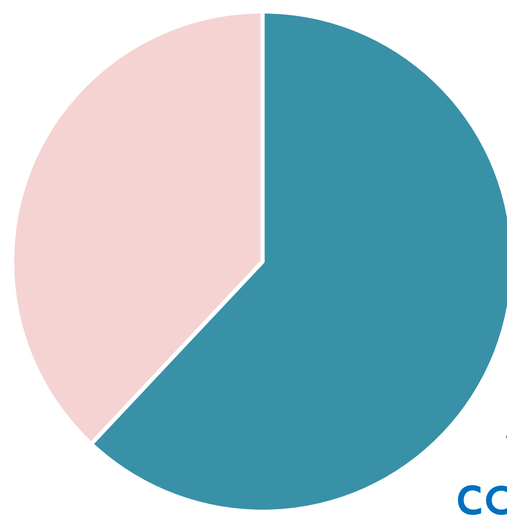
Comes with heavy efficiency costs

Conventional: apply C.S. to **all methods**

Problem of Context Sensitivity (C.S.)

Comes with heavy efficiency costs

Conventional: apply C.S. to **all methods**



Do **not** benefit from C.S.

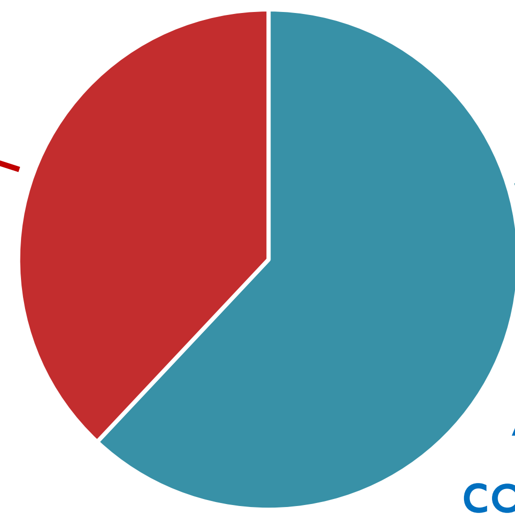
Analyzed for multiple contexts **redundantly**

Problem of Context Sensitivity (C.S.)

Comes with heavy efficiency costs

Conventional: apply C.S. to **all methods**

Benefit from C.S. (gain precision)



Do **not** benefit from C.S.

Precision-critical methods

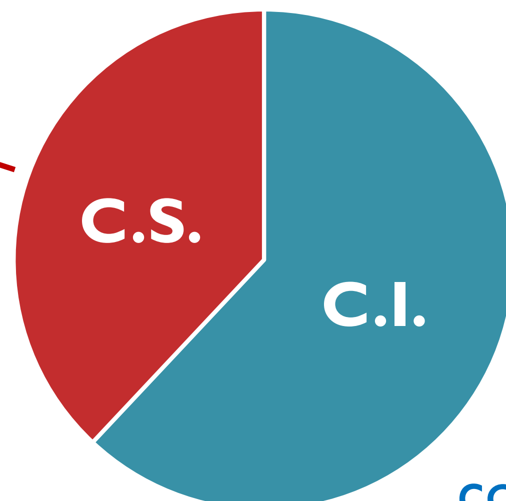
Analyzed for multiple contexts **redundantly**

Problem of Context Sensitivity (C.S.)

Comes with heavy efficiency costs

Benefit from C.S.
(gain precision)

Do **not** benefit from C.S.



Precision-critical methods

Analyzed for multiple contexts **redundantly**

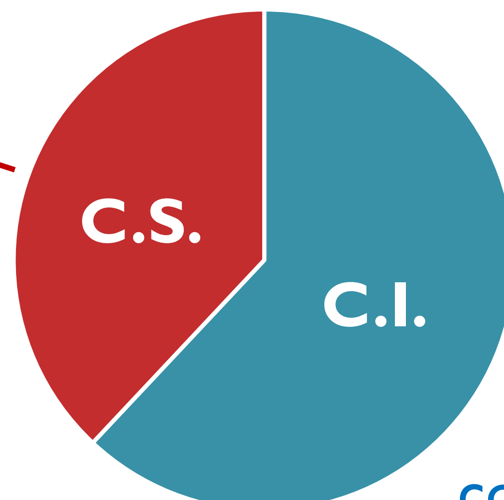
Problem of Context Sensitivity (C.S.)

Comes with heavy efficiency costs

Preserve precision
Improve efficiency
of C.S.

Benefit from C.S.
(gain precision)

Precision-critical methods



Do **not** benefit from C.S.

Analyzed for multiple contexts **redundantly**

Our Goal

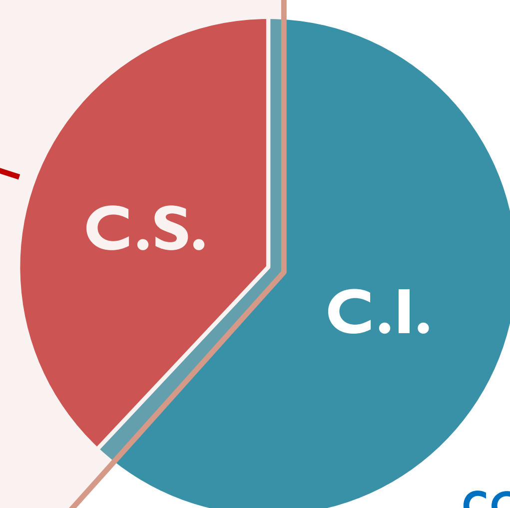
Identify **precision-critical methods**

Preserve precision
Improve efficiency

of C.S.

Benefit from C.S.
(gain precision)

Precision-critical methods



Do **not** benefit from C.S.

Analyzed for multiple contexts **redundantly**

Challenge

Still unclear where and how imprecision is introduced in a context-insensitive pointer analysis

When?	context-sensitive analysis	yield →	precision benefits
When?	omitting context sensitivity	introduce →	precision losses

Our Key Contribution

Classify source of imprecision into three general precision-loss patterns

- *Direct flow*
- *Wrapped flow*
- *Unwrapped flow*

Our Key Contribution

Classify source of imprecision into three general precision-loss patterns

- *Direct flow*
- *Wrapped flow*
- *Unwrapped flow*

account for ~99% of precision

Our Key Contribution

Classify source of imprecision into three general precision-loss patterns

- *Direct flow*
- *Wrapped flow*
- *Unwrapped flow*

} account for ~99% of precision

Recognize Three Flow Patterns ⇒ Identify Precision-Critical Methods

IN and OUT Methods

Given a class

- IN methods
  - One or more parameters
- OUT methods
  - non-`void` return types

IN and OUT Methods

Given a class

- IN methods
  - One or more parameters
- OUT methods
  - non-void return types

```
class Foo {
  C f;

  void setF(C p) {
    this.f = p;
  }

  C getF() {
    C r = this.f;
    return r;
  }

  void bar() {
    this.f = null;
  }
}
```

IN and OUT Methods

Given a class

- IN methods
  - One or more parameters
- OUT methods
  - non-void return types

IN

OUT

```
class Foo {
  C f;

  void setF(C p) {
    this.f = p;
  }

  C getF() {
    C r = this.f;
    return r;
  }

  void bar() {
    this.f = null;
  }
}
```

The Three General Flow Patterns

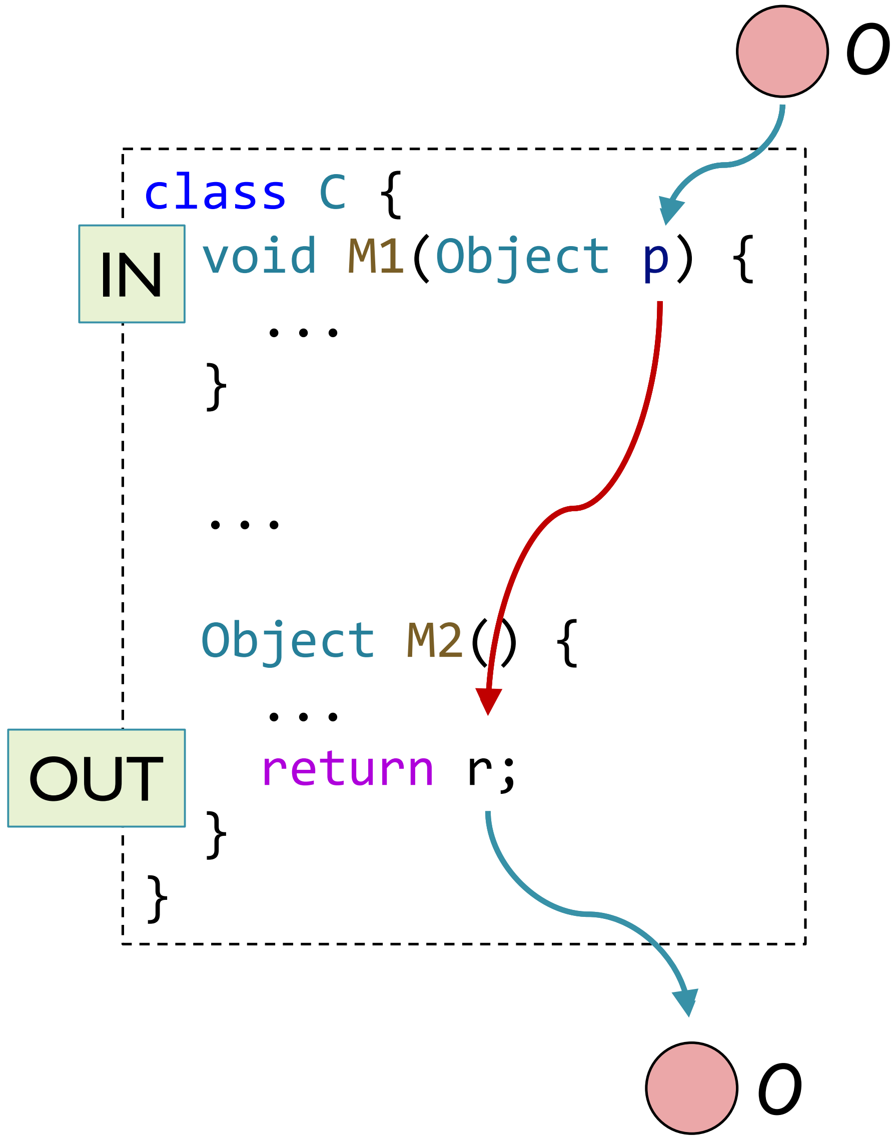
- Direct flow
- Wrapped flow
- Unwrapped flow

Identified by leveraging a context-insensitive pointer analysis (as pre-analysis)

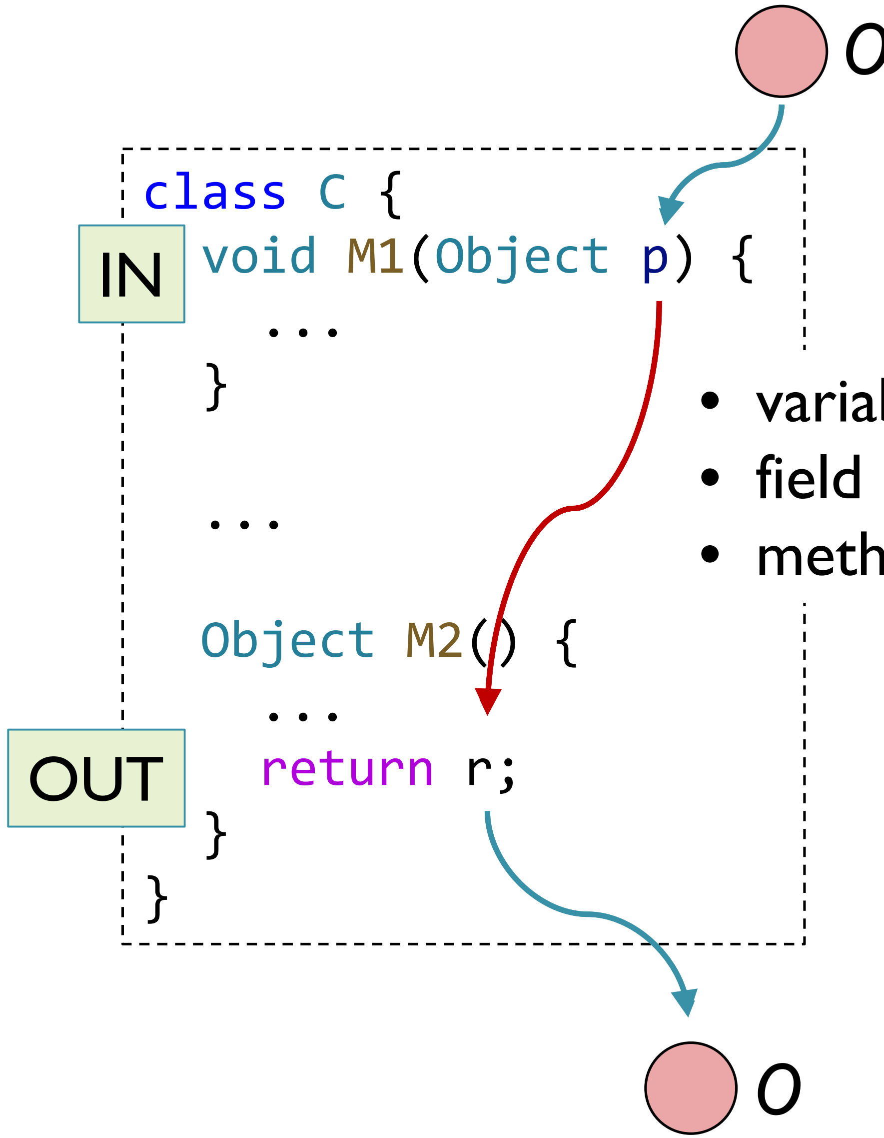
The Three General Flow Patterns

- Direct flow
- Wrapped flow
- Unwrapped flow

Direct Flow



Direct Flow



- variable assignments
- field load/store
- method calls/returns

Direct Flow

```
void set(Object p) {
  this.f = p;
}

Object get() {
  Object r = this.f;
  return r;
}
```

IN

OUT

```
class C {
  void M1(Object p) {
    ...
  }

  ...

  Object M2() {
    ...
    return r;
  }
}
```

o

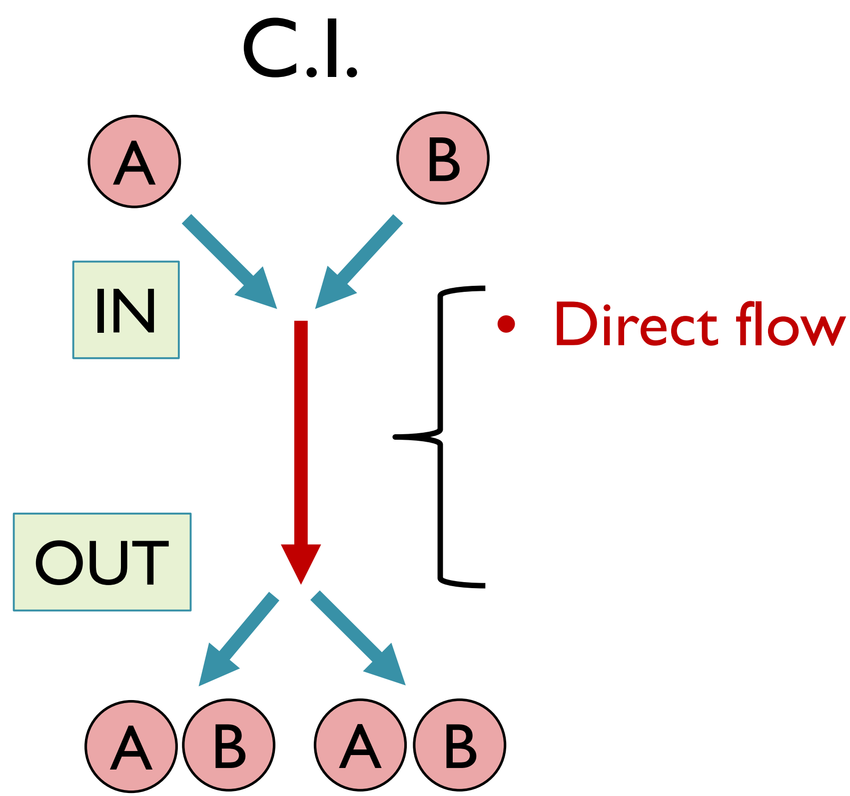
o

- variable assignments
- field load/store
- method calls/returns

Example: common setter & getter

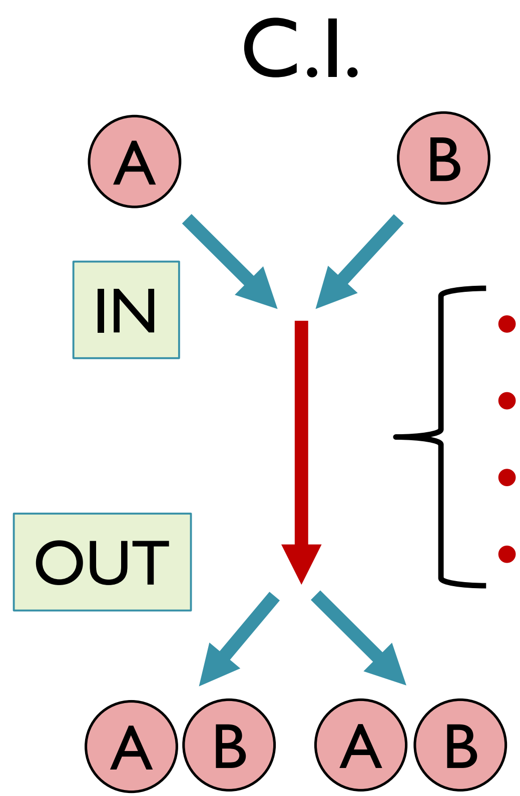
Key Insight: Causes of Imprecision

Flows: objects merge and propagate



Key Insight: Causes of Imprecision

Flows: objects merge and propagate

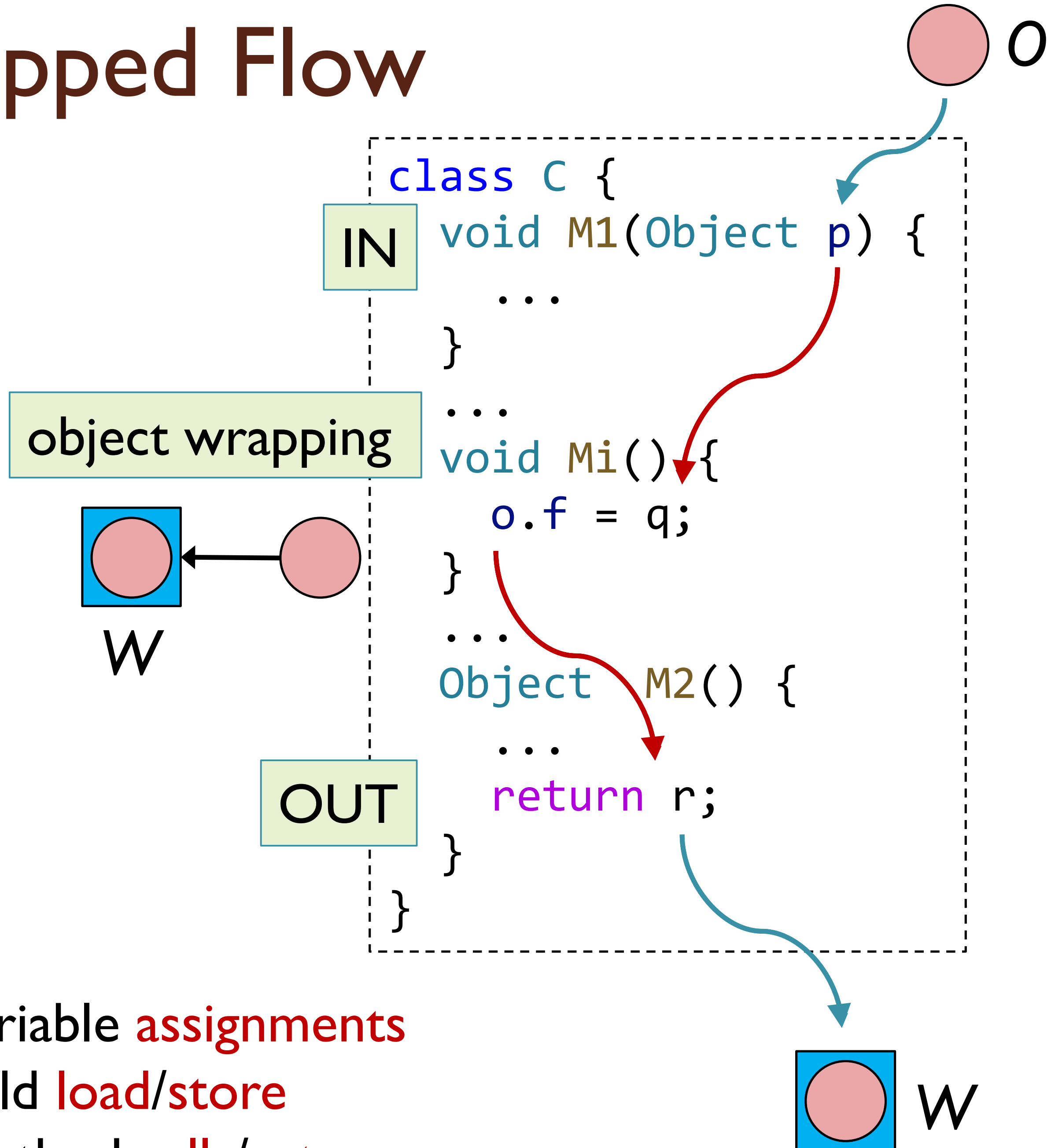


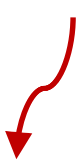
- Direct flow
- Wrapped flow
- Unwrapped flow
- Combinations

The Three General Flow Patterns

- Direct flow
- Wrapped flow
- Unwrapped flow

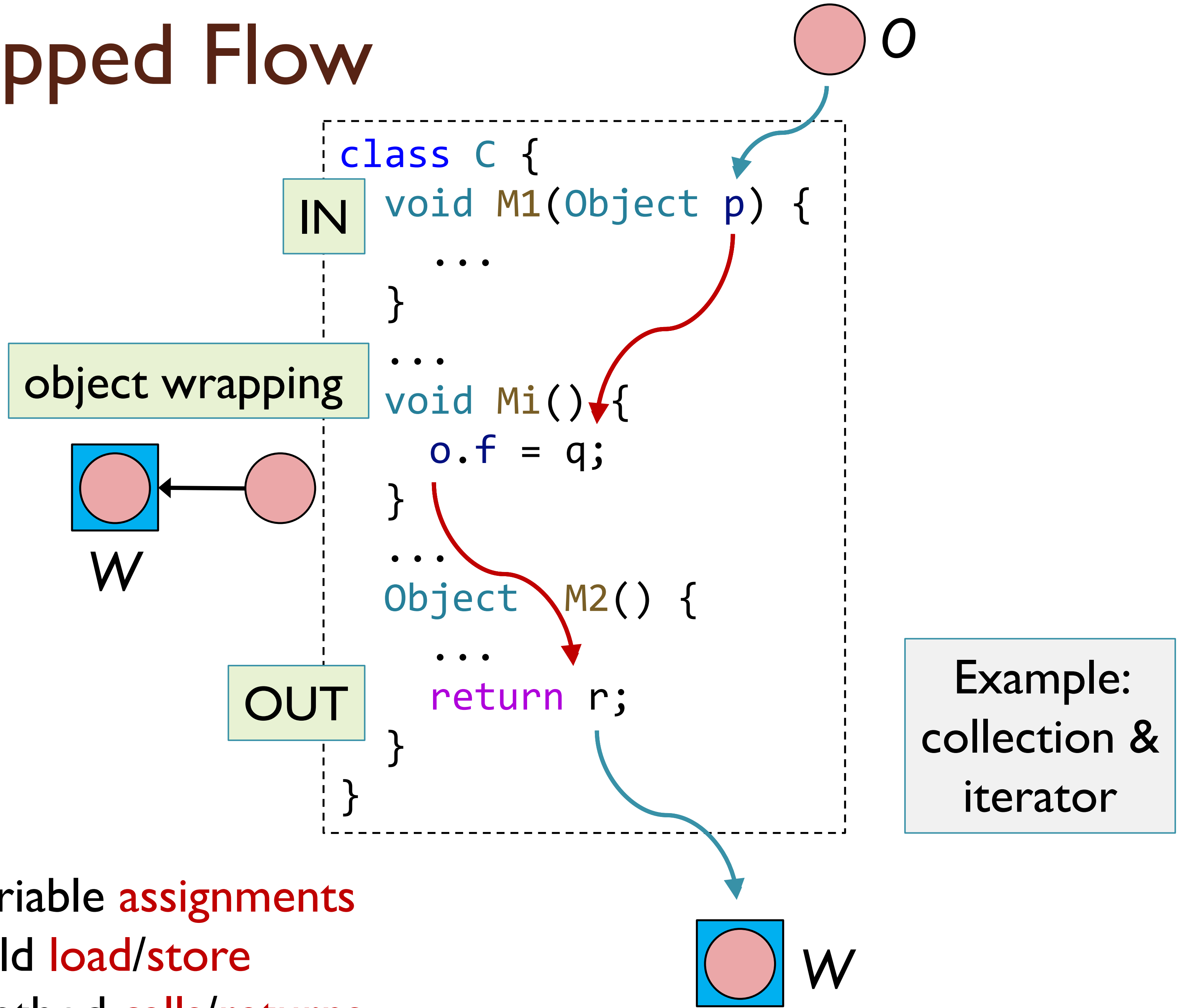
Wrapped Flow





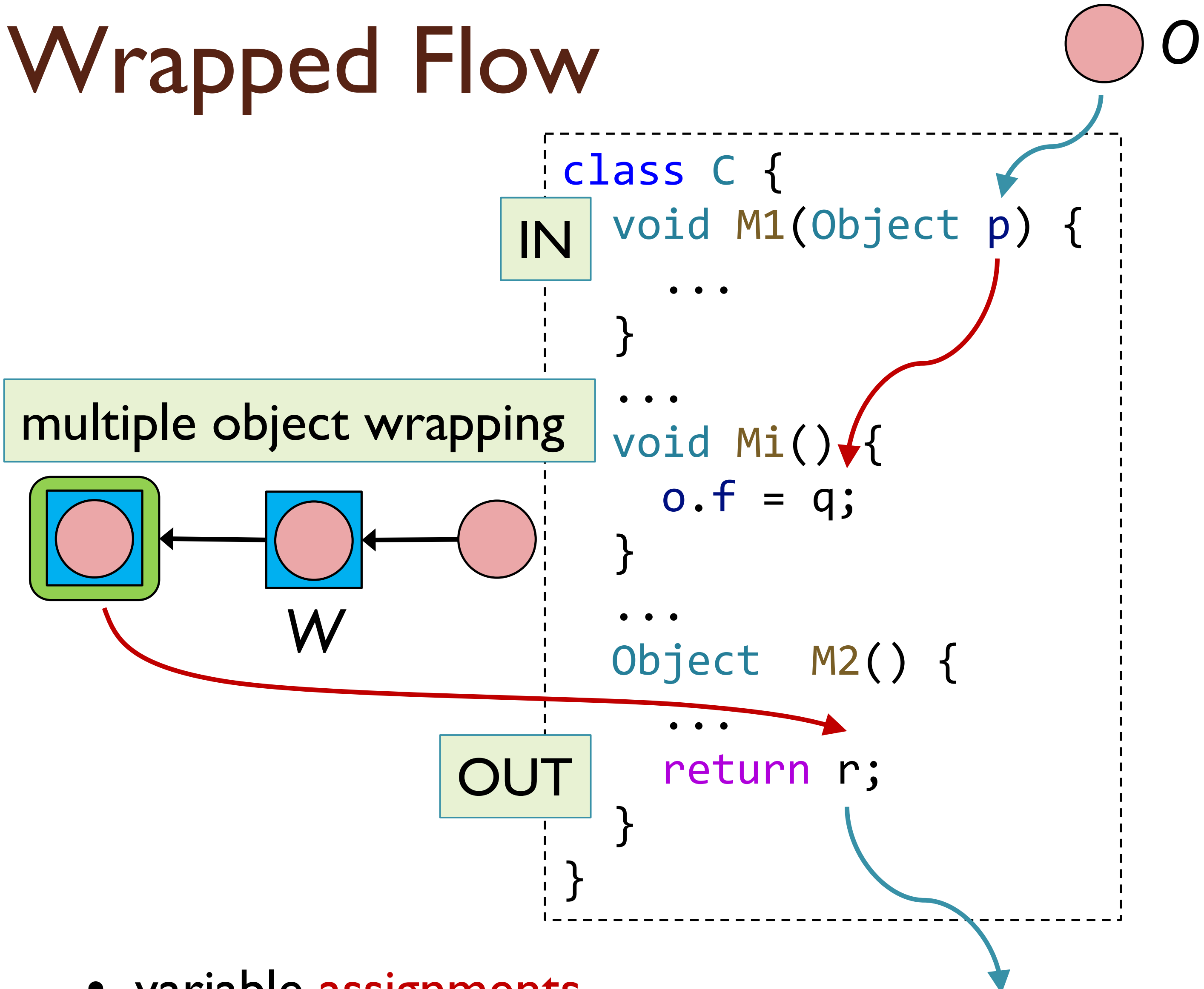
- variable assignments
- field load/store
- method calls/returns

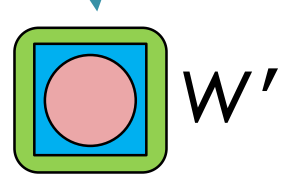
Wrapped Flow



- variable assignments
- field load/store
- method calls/returns

Wrapped Flow



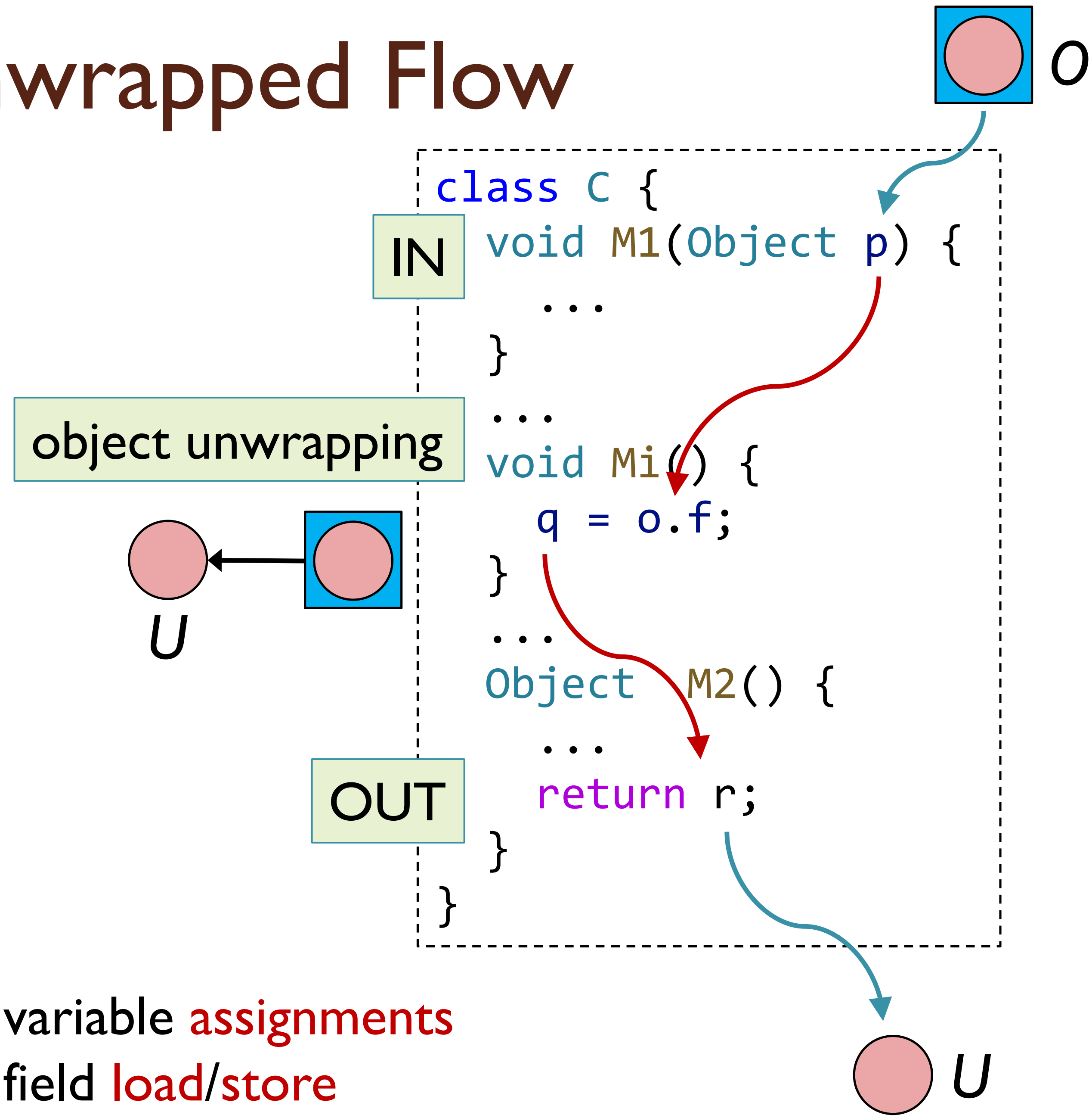


- variable assignments
- field load/store
- method calls/returns

The Three General Flow Patterns

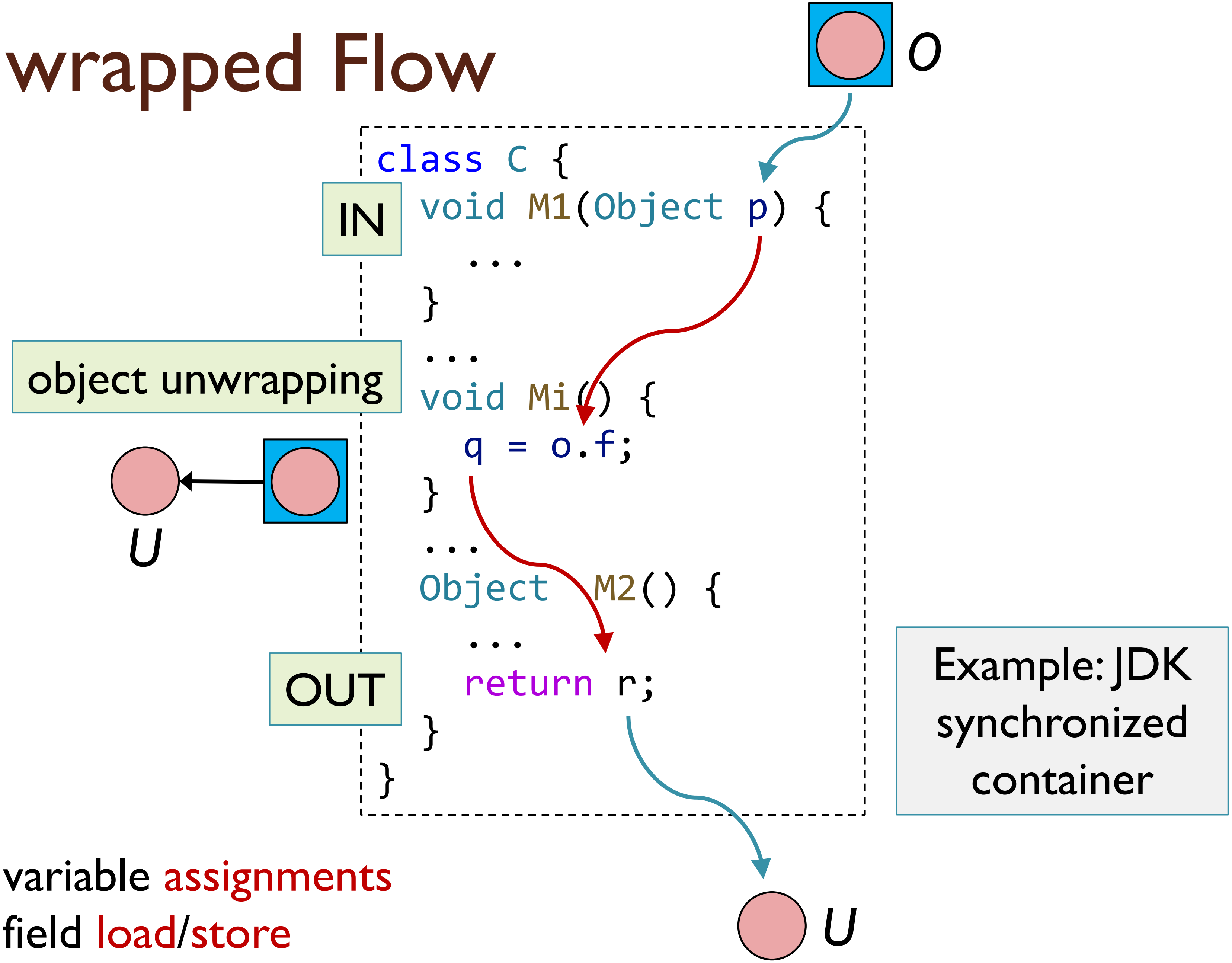
- Direct flow
- Wrapped flow
- Unwrapped flow

Unwrapped Flow



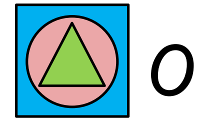
- variable assignments
- field load/store
- method calls/returns

Unwrapped Flow



- variable assignments
- field load/store
- method calls/returns

Unwrapped Flow

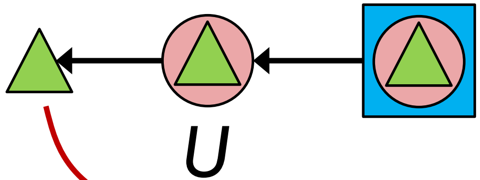


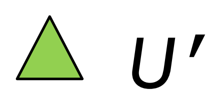
IN

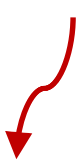
```
class C {
  void M1(Object p) {
    ...
  }
  ...
  void Mi() {
    q = o.f;
  }
  ...
  Object  M2() {
    ...
    return r;
  }
}
```

OUT

multiple object unwrapping



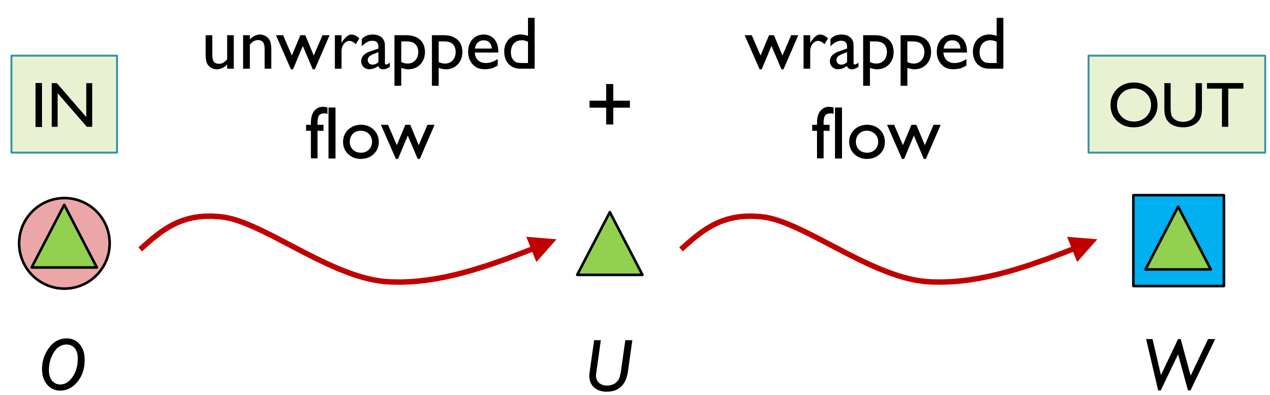




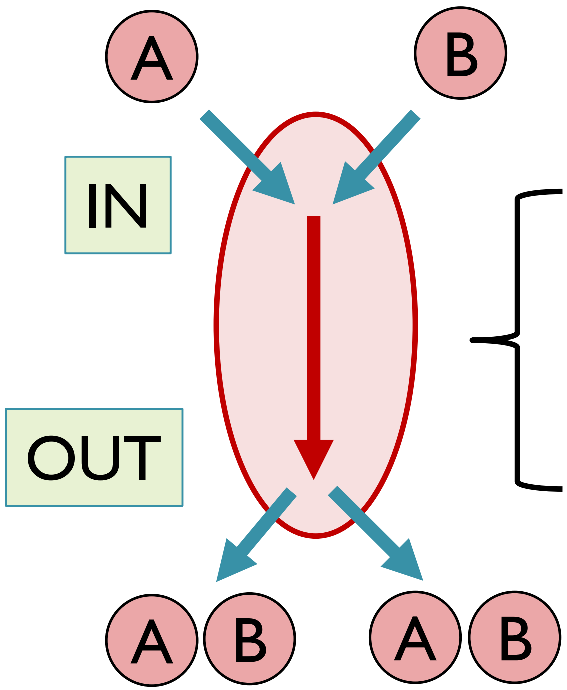
- variable assignments
- field load/store
- method calls/returns

Combinations of Three General Flows

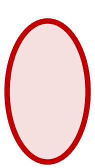
The direct, wrapped and unwrapped flows can be combined, e.g.,



C.I.

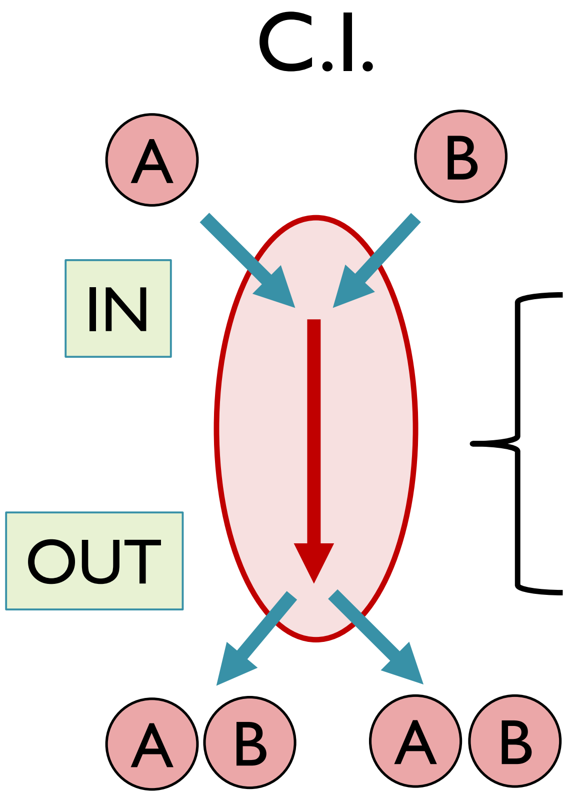


- Direct flow
- Wrapped flow
- Unwrapped flow
- Combinations

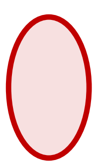


Precision-critical methods:
the methods involved in the flows

Identify precision-critical methods



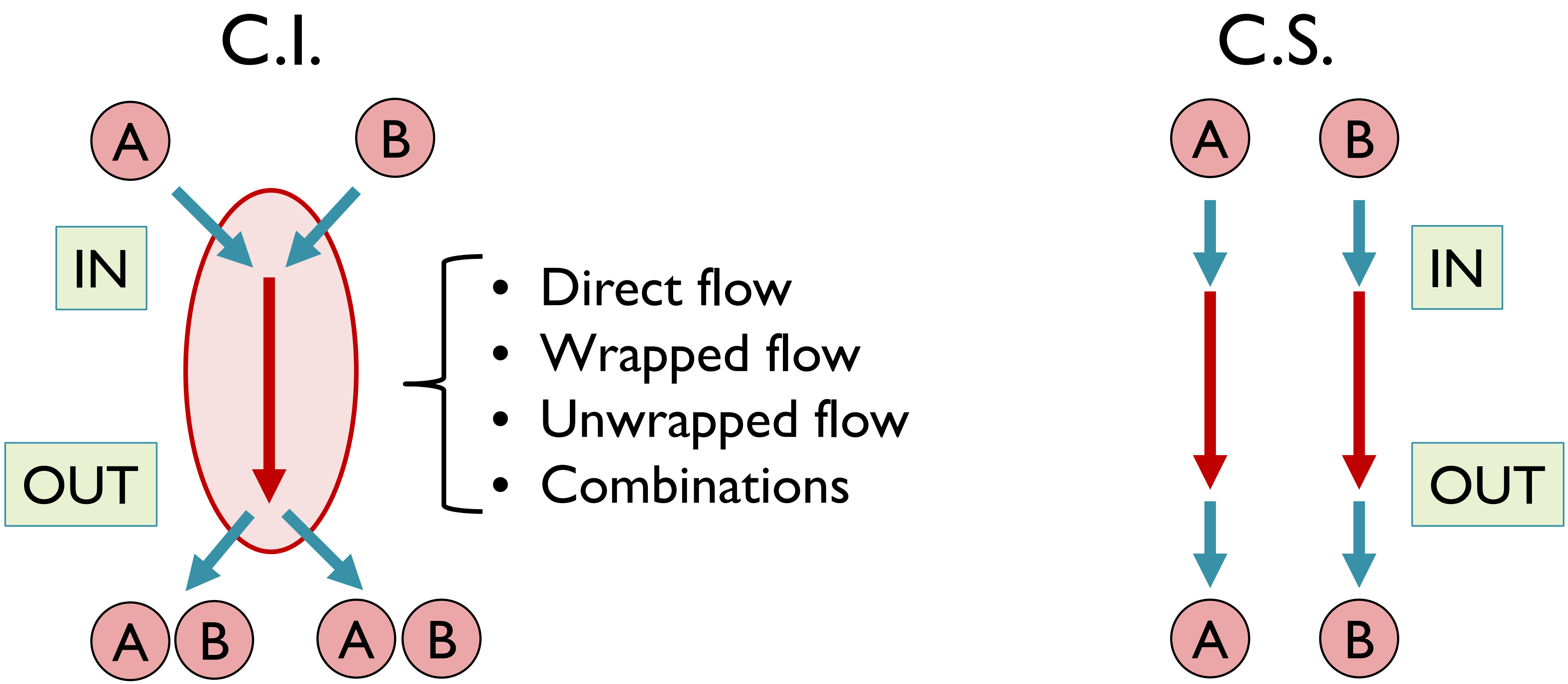
- Direct flow
- Wrapped flow
- Unwrapped flow
- Combinations



Precision-critical methods:
the methods involved in the flows

Identify precision-critical methods

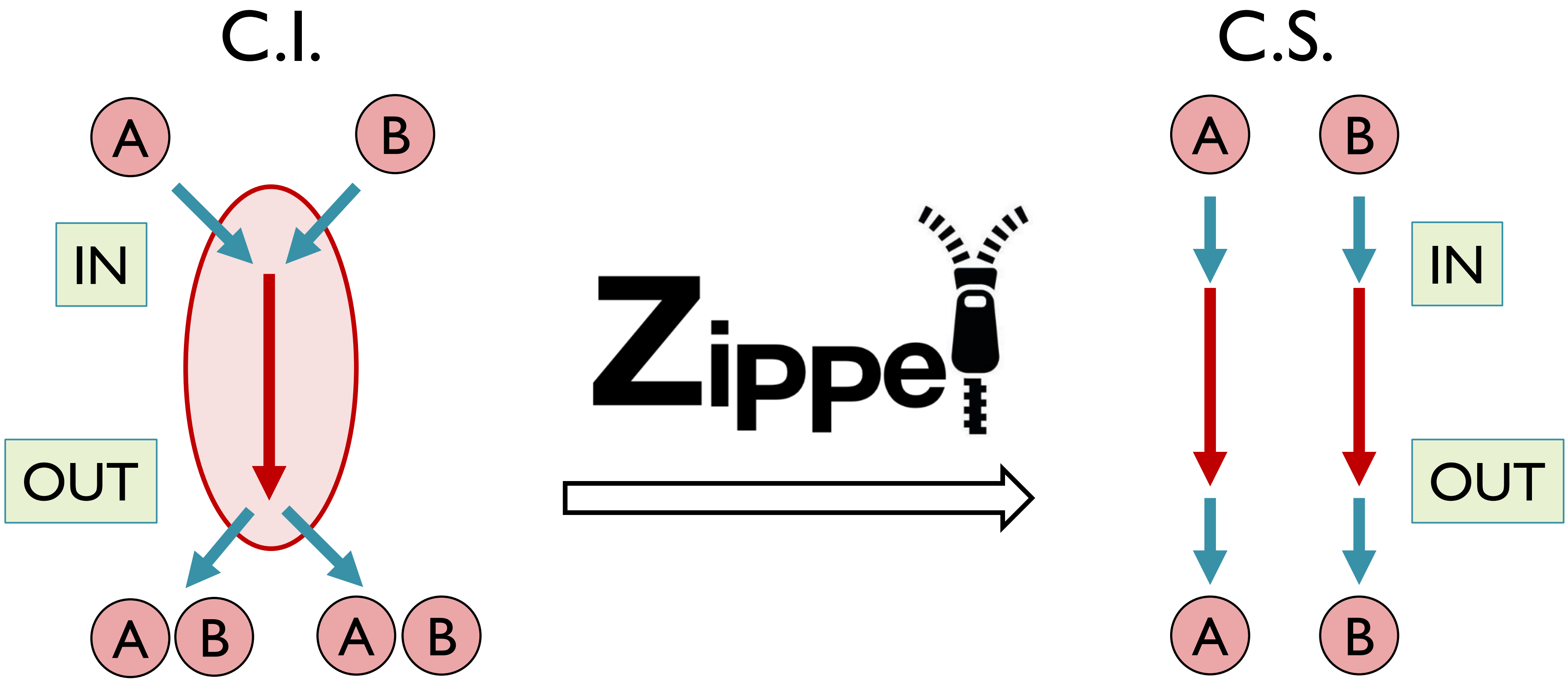
Apply C.S. only to



Precision-critical methods:
the methods involved in the flows

Identify precision-critical methods

Apply C.S. only to



Precision-critical methods:

the methods involved in the flows

How to Analyze Flow Patterns?

We propose precision flow graph (PFG)

- expresses direct, wrapped, unwrapped flows, and their combinations, in an uniform way

How to Analyze Flow Patterns?

We propose precision flow graph (PFG)

expresses direct, wrapped, unwrapped flows, and their combinations, in an uniform way

Flows in Program ⟺ Paths in PFG

Precision Flow Graph (PFG)

Flows in Program ⟺ Paths in PFG

- Statically over-approximates all the general flows and their combinations
- Based on the results of context-insensitive pointer analysis (pre-analysis)

How to Analyze Flow Patterns?

We propose precision flow graph (PFG)

expresses direct, wrapped, unwrapped flows, and their combinations, in an uniform way

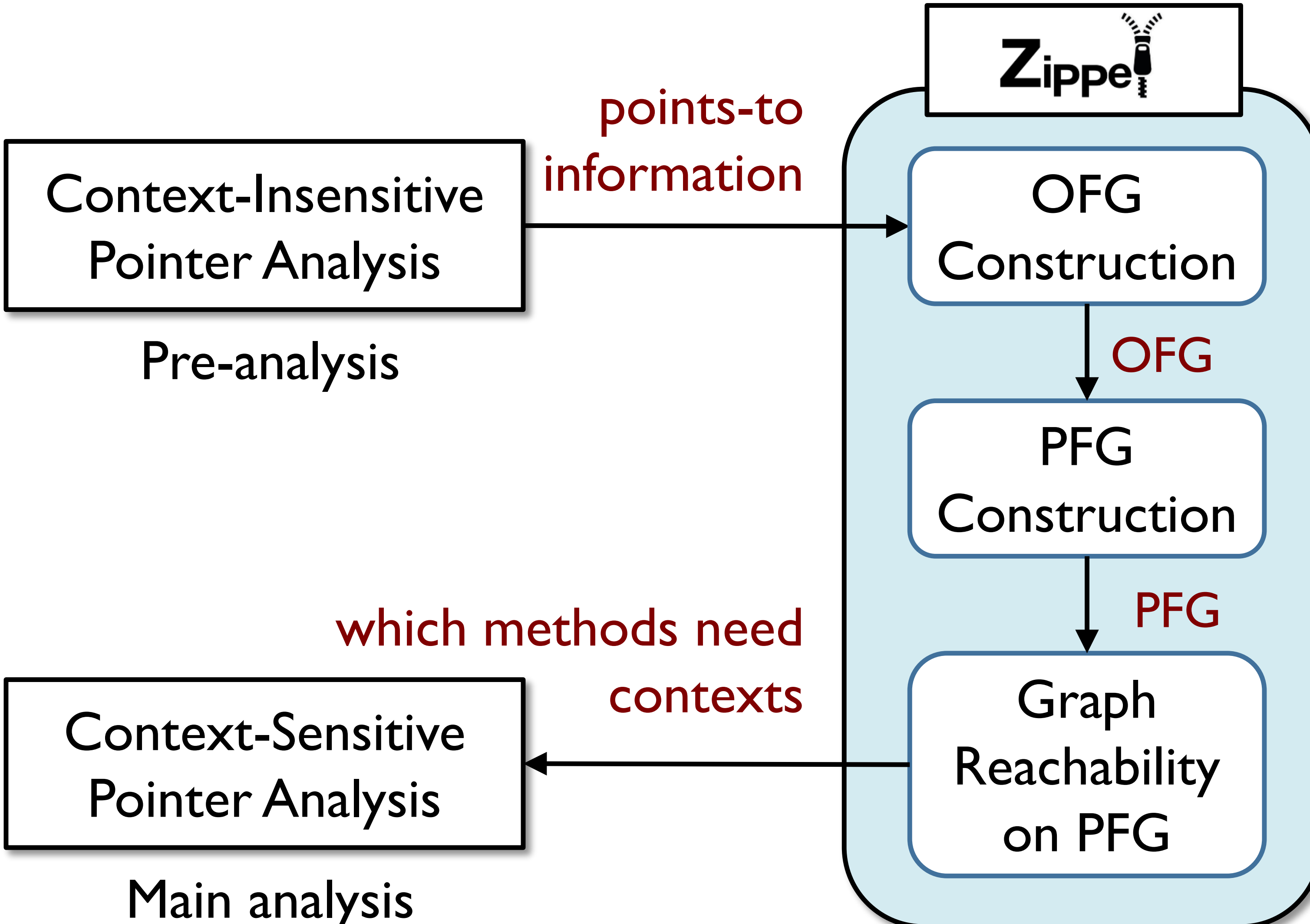
Flows in Program ⟺ Paths in PFG

Methods Involved in the Flows ⟺ Simple Graph Reachability on PFG

i.e., precision-critical methods

from IN to OUT methods

Overview



Zipper
points-to information
Context-Insensitive Pointer Analysis
Pre-analysis
OFG Construction
OFG
PFG Construction
PFG
Graph Reachability on PFG
which methods need contexts
Context-Sensitive Pointer Analysis
Main analysis

Implementation

- Written in Java (core: ~1500 LOC)
- Integrated with DOOP
- Can also be easily integrated with other pointer analysis frameworks
- Open source: http://www.brics.dk/zipper/

Evaluation

- Compared to conventional context-sensitive analysis, can ZIPPER-guided analysis
  - preserve precision?
  - improve efficiency?

Evaluation

- Compared to conventional context-sensitive analysis, can ZIPPER-guided analysis
  - preserve precision?
  - improve efficiency?

- Context sensitivity: 2-object-sensitivity (2obj)
  - Most practical high-precision pointer analysis
  - Widely adopted (research papers and analysis frameworks)

Evaluation

- Compared to conventional context-sensitive analysis, can ZIPPER-guided analysis
  - preserve precision?
  - improve efficiency?

- Context sensitivity: 2-object-sensitivity (2obj)
  - Conventional: applies 2obj to all methods
  - ZIPPER-guided: applies 2obj to only precision-critical methods selected by ZIPPER

Evaluation - Analyzed Programs

10 large Java programs

- 5 popular real-world applications

- 5 DaCapo benchmarks

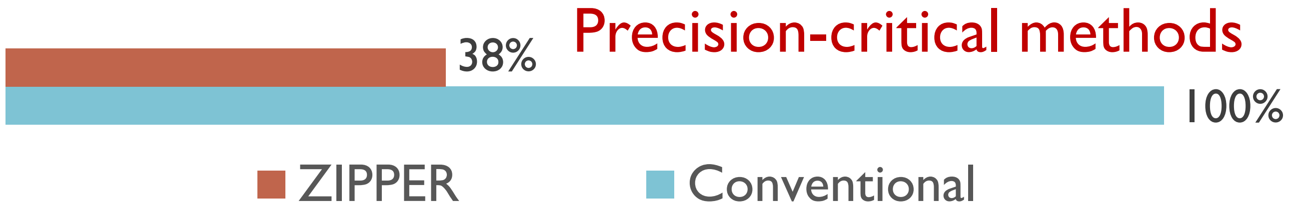
Evaluation - Clients

- May-fail casting
- De-virtualization
- Method reachability
- Call graph construction

Widely-used clients to evaluate pointer analysis's precision
e.g., PLDI'17, OOPSLA'17, PLDI'14, PLDI'13, POPL'11, OOPSLA'09 …

Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

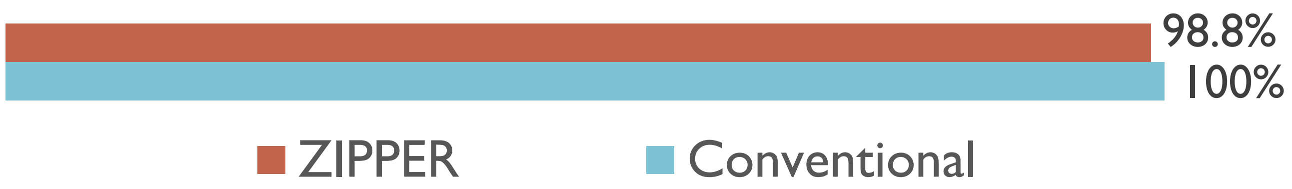


Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

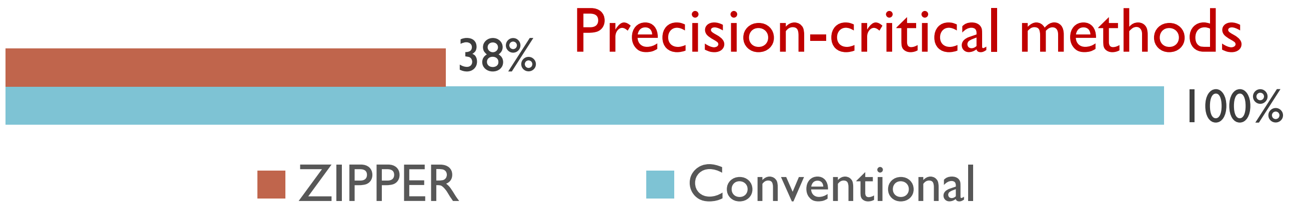
Precision-critical methods

Precision



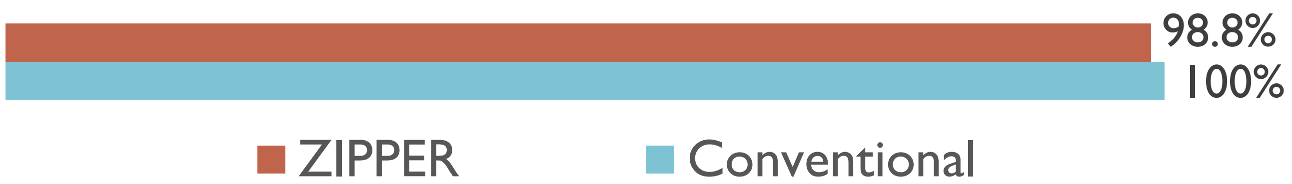
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)



Precision

C.I. 64.5%



Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

Precision-critical methods

Precision

C.I. 64.5%

Preserve precision

Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

Precision-critical methods

Precision

C.I. 64.5%

Preserve precision

Analysis Time

ZIPPER: 3.4X of speedup (up to 9.2X)

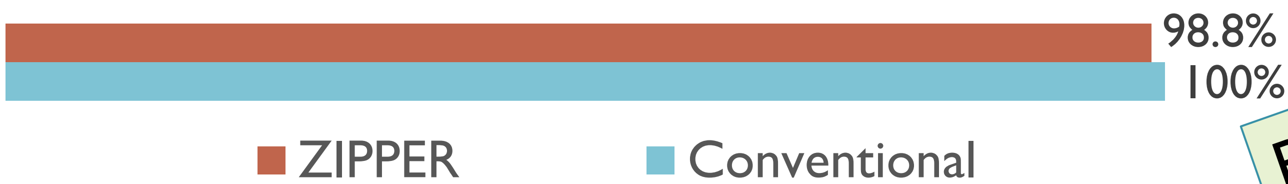
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

Precision-critical methods

Precision

C.I. 64.5%



Preserve precision ✔

Analysis Time

ZIPPER: 3.4X of speedup (up to 9.2X)

Improve efficiency ✔

Conclusion

Zipper

- Direct, wrapped, and unwrapped flows
  - explain where and how most imprecision is introduced in context insensitivity
- Precision flow graph
  - concisely models the above flows
- Implementation (http://www.brics.dk/zipper/)
  - effectively identifies precision-critical methods
- Evaluation
  - preserves essentially all of the precision
  - improves efficiency significantly

The Parameter-Out Flow Case

```
void m(A input, B output) {
  output.field = input;
}
```

```
m(a, b);         // rare
b.setField(a); // common
```

Potential of ZIPPER

bloat	Time(s)	#fail-cast	#poly-call	#reach-mtd	#call-edge
Conventional	3128	1193	1427	8470	53143
Zipper	2704	1224	1449	8486	53289
Zipper*	**52**	1310	1511	8538	54049

ZIPPER*: tracks flows from an IN method only if its flowing-in objects have too many (>50) different types

Identify highly precision-critical methods

More heuristics and precision-efficiency trade-offs can be developed on top of ZIPPER