Precision-Guided Context Sensitivity for Pointer Analysis

Yue Li, Tian Tan, Anders Møller, Yannis Smaragdakis

OOPSLA 2018

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

<table>
<thead>
<tr>
<th>Clients</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Security analysis</td>
<td>Soot, DroidSafe</td>
</tr>
<tr>
<td>• Bug detection</td>
<td>Doop, Chord</td>
</tr>
<tr>
<td>• Compiler optimization</td>
<td></td>
</tr>
<tr>
<td>• Program verification</td>
<td></td>
</tr>
<tr>
<td>• Program understanding</td>
<td></td>
</tr>
<tr>
<td>• …</td>
<td>…</td>
</tr>
</tbody>
</table>
# Uses of Pointer Analysis

<table>
<thead>
<tr>
<th>Clients</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Security analysis</td>
<td><img src="#" alt="Soot" /> <img src="#" alt="DroidSafe" /></td>
</tr>
<tr>
<td>• Bug detection</td>
<td><img src="#" alt="DOOP" /> <img src="#" alt="Chord" /></td>
</tr>
<tr>
<td>• Compiler optimization</td>
<td><img src="#" alt="WALA" /> <img src="#" alt="T. J. Watson Libraries for Analysis" /></td>
</tr>
<tr>
<td>• Program verification</td>
<td></td>
</tr>
<tr>
<td>• Program understanding</td>
<td></td>
</tr>
<tr>
<td>• ...</td>
<td></td>
</tr>
</tbody>
</table>

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");
}

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
<tr>
<td>b1</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
<tr>
<td>b2</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
</tbody>
</table>

Context-Insensitivity
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");
}

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>&quot;s1&quot;, &quot;s2&quot;</td>
</tr>
<tr>
<td>b1</td>
<td>&quot;s1&quot;, &quot;s2&quot;</td>
</tr>
<tr>
<td>b2</td>
<td>&quot;s1&quot;, &quot;s2&quot;</td>
</tr>
</tbody>
</table>

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");
}

<table>
<thead>
<tr>
<th>Context</th>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A/1]</td>
<td>s</td>
<td>&quot;s1&quot;</td>
</tr>
<tr>
<td>[A/2]</td>
<td>s</td>
<td>&quot;s2&quot;</td>
</tr>
<tr>
<td>[]</td>
<td>b1</td>
<td>&quot;s1&quot;</td>
</tr>
<tr>
<td>[]</td>
<td>b2</td>
<td>&quot;s2&quot;</td>
</tr>
</tbody>
</table>

I-Object-Sensitivity

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
<tr>
<td>b1</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
<tr>
<td>b2</td>
<td>&quot;s1&quot;,&quot;s2&quot;</td>
</tr>
</tbody>
</table>

Context-Insensitivity
Context Sensitivity

Widely adopted by static analysis frameworks for **OO programs**

[Images of Chord, Doop, FlowDroid, TAJS, Soot, and DroidSafe]
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.

Analyzing 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)
Precision-critical methods

Do not benefit from C.S.

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

- Benefit from C.S. (gain precision)
- Do not benefit from C.S.
- Analyzed for multiple contexts redundantly

Preserve precision
Improve efficiency

Precision-critical methods
Our Goal

Identify **precision-critical methods**

- Benefit from C.S. (gain precision)
- Precision-critical methods
- Do not benefit from C.S.
- Analyzed for multiple contexts **redundantly**
- Preserve precision
- Improve efficiency of C.S.
Challenge

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

These three flow patterns account for ~99% of precision loss.

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

```java
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

```java
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
class C {
    void M1(Object p) {
        ...
    }
    ...
    Object M2() {
        ...
        return r;
    }
}
class C {
    void M1(Object p) {
        ...
    }
    Object M2() {
        ...
        return r;
    }
}

Direct Flow

IN

OUT

• variable assignments
• field load/store
• method calls/returns
void set(Object p) {
    this.f = p;
}

class C {
    void M1(Object p) {
        ...
    }
    ...
    Object M2() {
        ...
        Object r = this.f;
        return r;
    }
}

Object get() {
    Object r = this.f;
    return r;
}
Key Insight: Causes of Imprecision

Flows: objects merge and propagate

C.I.

• Direct flow
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

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

- Variable assignments
- Field load/store
- Method calls/returns
Wrapped Flow

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

Example: collection & iterator

• variable assignments
• field load/store
• method calls/returns
Wrapped Flow

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

multiple object wrapping

• variable assignments
• field load/store
• method calls/returns
The Three General Flow Patterns

- Direct flow
- Wrapped flow
- Unwrapped flow
Unwrapped Flow

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

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

Example: JDK synchronized container

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

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

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.,
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

**C.I.**

- Direct flow
- Wrapped flow
- Unwrapped flow
- Combinations

**C.S.**

IN

OUT

A

B

A

B

A

B

IN

OUT

A

B
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.
Precision Flow Graph (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

Context-Insensitive Pointer Analysis

Pre-analysis

Context-Sensitive Pointer Analysis

Main analysis

points-to information

which methods need contexts

OFG Construction

PFG Construction

Graph Reachability on PFG

Zippe
Implementation

- Written in Java (core: ~1500 LOC)
- Integrated with [Doop](http://www.brics.dk/zipper/)
- Can also be easily integrated with other pointer analysis frameworks
- Open source: [http://www.brics.dk/zipper/](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
  - batik
  - checkstyle
  - SunFlow
  - JPC
  - FindBugs
- 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)

- **ZIPPER**
  - 38%
- **Conventional**
  - 100%

**Precision-critical methods**
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

- Precision-critical methods
  - ZIPPER: 38%
  - Conventional: 100%

Precision

- ZIPPER: 98.8%
- Conventional: 100%
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

<table>
<thead>
<tr>
<th>Precision-critical methods</th>
<th>ZIPPER</th>
<th>Conventional</th>
</tr>
</thead>
<tbody>
<tr>
<td>38%</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

Precision

<table>
<thead>
<tr>
<th>ZIPPER</th>
<th>Conventional</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.8%</td>
<td>100%</td>
</tr>
</tbody>
</table>

C.I. 64.5%
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

- ZIPPER: 38%
- Conventional: 100%

Precision-critical methods

Precision

- ZIPPER: 98.8%
- Conventional: 100%

C.I. 64.5%

Preserve precision
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

- Precision-critical methods
  - ZIPPER: 38%
  - Conventional: 100%

- Precision
  - ZIPPER: 98.8%
  - Conventional: 100%
  
- Analysis Time
  - ZIPPER: 3.4X of speedup (up to 9.2X)
  - Conventional: 

C.I. 64.5%

Preserve precision

Precise results: 38% of precision-critical methods are analyzed context-sensitively. ZIPPER improves precision by up to 9.2X compared to conventional methods, while maintaining 100% precision and preserving context sensitivity.
Results: ZIPPER vs. Conventional

Methods Analyzed Context-Sensitively (2obj)

<table>
<thead>
<tr>
<th>ZIPPER</th>
<th>Conventional</th>
</tr>
</thead>
<tbody>
<tr>
<td>38%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Precision-critical methods

Precision

<table>
<thead>
<tr>
<th>ZIPPER</th>
<th>Conventional</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.8%</td>
<td>100%</td>
</tr>
</tbody>
</table>

C.I. 64.5%

Preserve precision

Analysis Time

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

ZIPPER

Conventional

Improve efficiency
Conclusion

- 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

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

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

<table>
<thead>
<tr>
<th>bloat</th>
<th>Time(s)</th>
<th>#fail-cast</th>
<th>#poly-call</th>
<th>#reach-mtd</th>
<th>#call-edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>3128</td>
<td>1193</td>
<td>1427</td>
<td>8470</td>
<td>53143</td>
</tr>
<tr>
<td>Zipper</td>
<td>2704</td>
<td>1224</td>
<td>1449</td>
<td>8486</td>
<td>53289</td>
</tr>
<tr>
<td>Zipper*</td>
<td>52</td>
<td>1310</td>
<td>1511</td>
<td>8538</td>
<td>54049</td>
</tr>
</tbody>
</table>

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.