Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata

Tian Tan, Yue Li and Jingling Xue

PLDI 2017
June, 2017
A New Points-to Analysis Technique for Object-Oriented Programs
Points-to Analysis

- Determines
  - “which objects a variable can point to?”
# Uses of Points-to Analysis

<table>
<thead>
<tr>
<th>Clients</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security analysis</td>
<td>Soot</td>
</tr>
<tr>
<td>Bug detection</td>
<td>DroidSafe</td>
</tr>
<tr>
<td>Compiler optimization</td>
<td>Doop</td>
</tr>
<tr>
<td>Program verification</td>
<td>Chord</td>
</tr>
<tr>
<td>Program understanding</td>
<td>WALA</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# Uses of Points-to Analysis

<table>
<thead>
<tr>
<th>Clients</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Security analysis</td>
<td>• Soot</td>
</tr>
<tr>
<td>• Bug detection</td>
<td>• DroidSafe</td>
</tr>
<tr>
<td>• Compiler optimization</td>
<td>• DOOP</td>
</tr>
<tr>
<td>• Program verification</td>
<td>• Chord</td>
</tr>
<tr>
<td>• Program understanding</td>
<td>• WALA</td>
</tr>
<tr>
<td>• ...</td>
<td>• ...</td>
</tr>
</tbody>
</table>

Call Graph
Existing Call Graph Construction

- On-the-fly construction (run with points-to analysis)
  - Precise
  - Inefficient
Existing Call Graph Construction

- On-the-fly construction (run with points-to analysis)
  - Precise
  - Inefficient

- 3-object-sensitive points-to analysis
  - Very precise
  - Adopted by, e.g., Doop, DroidSafe, Chord
3-Object-Sensitive Points-to Analysis

- Analyze Java programs
  - Intel Xeon E5 3.70GHz, 128GB of memory
  - Time budget: 5 hours (18000 secs)
3-Object-Sensitive Points-to Analysis

- Analyze Java programs
  - Intel Xeon E5 3.70GHz, 128GB of memory
  - Time budget: 5 hours (18000 secs)

![Graph showing analysis time for pmd and findbugs]

- **findbugs**
  - Analysis time: 14469 seconds (4 hours)
  - Unscalable (>, 5 hours)

- **pmd**
  - Analysis time: 14469 seconds (4 hours)
  - Unscalable (>, 5 hours)
Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
- Model data-flow
Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
  - Context-sensitivity
    - Call-site-sensitivity (PLDI’04, PLDI’06)
    - Object-sensitivity (ISSTA’02, TOSEM’05, SAS’16)
    - Type-sensitivity (POPL’11)
    - ...

- Model data-flow
Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
  - Context-sensitivity
    - Call-site-sensitivity (PLDI’04, PLDI’06)
    - Object-sensitivity (ISSTA’02, TOSEM’05, SAS’16)
    - Type-sensitivity (POPL’11)
    - ...

- Model data-flow
  - Heap abstraction
    - Allocation-site abstraction
    - Type-based abstraction
    - ...

Two Mainstreams of Points-to Analysis Techniques

- **Model control-flow**
  - Context-sensitivity
    - Call-site-sensitivity (PLDI’04, PLDI’06)
    - Object-sensitivity (ISSTA’02, TOSEM’05, SAS’16)
    - Type-sensitivity (POPL’11)
    - ...

- **Model data-flow**
  - **Heap abstraction**
    - Allocation-site abstraction
    - Type-based abstraction
    - ...


Heap Abstraction

Dynamic execution

Infinite-size heap

abstracted or partitioned

Static analysis

Finite (abstract) objects
Allocation-Site Abstraction

- One object per allocation site

1. A a1 = new A();
2. A a2 = new A();
3. B b = new B();
Allocation-Site Abstraction

- One object per allocation site

1. \( A \ a1 = \text{new} \ A(); \)
2. \( A \ a2 = \text{new} \ A(); \)
3. \( B \ b = \text{new} \ B(); \)
Allocation-Site Abstraction

- One object per allocation site
  - Adopted by all mainstream points-to analyses

```java
1  A a1 = new A();
2  A a2 = new A();
3  B b = new B();
```
Allocation-Site Abstraction

- **Over-partition** for call graph construction

```java
void foo(Object o) {
    o.toString();
}
```

```java
1  A a1 = new A();
2  A a2 = new A();
3  foo(a1);
4  foo(a2);
```

```
A::toString()
```
Allocation-Site Abstraction

- **Over-partition for type-dependent clients**
  - Call graph construction
  - Devirtualization
  - May-fail casting
  - ...

1. `A a1 = new A();`
2. `A a2 = new A();`
3. `foo(a1);`
4. `foo(a2);`

```java
void foo(Object o) {
    o.toString();
    A a = (A) o;
}
```
Type-Based Abstraction

- One object per type

```java
1  A a1 = new A();
2  A a2 = new A();
3  B b = new B();
```
Type-Based Abstraction

- One object per type

```java
1  A a1 = new A();
2  A a2 = new A();
3  B b = new B();
```
Type-Based Abstraction

- Precision loss for type-dependent clients

```java
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();
a1.f = b;
a2.f = c;
Object o = a1.f;
o.toString();
```
Type-Based Abstraction

- Precision loss for type-dependent clients

```
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();
a1.f = b;
a2.f = c;
Object o = a1.f;
o.toString();
```
Type-Based Abstraction

- Precision loss for type-dependent clients

```java
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();

a1.f = b;
a2.f = c;

Object o = a1.f;
o.toString();
```
Type-Based Abstraction

- Precision loss for type-dependent clients

```java
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();
a1.f = b;
a2.f = c;
Object o = a1.f;
o.toString();
```

Object graph:

```
A --|-> B
  |    |
  v    v
O --|-> C
```

Methods called:

```
B::toString()
C::toString()
```
Type-Based Abstraction

- Precision loss for type-dependent clients

```java
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();
a1.f = b;
a2.f = c;
Object o = a1.f;
o.toString();
```

**False positive**
Our Goal:
Improve Efficiency
Preserve Precision
**MAHJONG: A New Heap Abstraction**

**Improve Efficiency**

### Analysis Time (sec.)

- **pmd**
  - MAHJONG: 128 sec.
  - Allocation-site abstraction: 14469 sec. (4 fours)

- **findbugs**
  - MAHJONG: 524 sec.
  - Allocation-site abstraction: Unscalable (> 5 hours)

*Adopted by all mainstream points-to analyses*
MAHJONG: A New Heap Abstraction

**Analysis Time (sec.)**

<table>
<thead>
<tr>
<th>Tool</th>
<th>MAHJONG</th>
<th>Allocation-site abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmd</td>
<td>128</td>
<td>14469 (4 fours)</td>
</tr>
<tr>
<td>findbugs</td>
<td>524</td>
<td>Unscalable (&gt; 5 hours)</td>
</tr>
</tbody>
</table>

**Improve Efficiency**

**#call graph edges**

<table>
<thead>
<tr>
<th>Tool</th>
<th>MAHJONG</th>
<th>Allocation-site abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmd</td>
<td>44016</td>
<td>44004</td>
</tr>
</tbody>
</table>

**Preserve Precision**

Adopted by all mainstream points-to analyses
MAHJONG: A New Heap Abstraction

**Improve Efficiency**

- Analysis Time (sec.):
  - pmd: 128 (MAHJONG) vs. 14469 (Allocation-site abstraction, 4 fours)
  - findbugs: 524 (MAHJONG) vs. Unscalable (> 5 hours)

**Preserve Precision**

- #call graph edges:
  - pmd: 44016 (MAHJONG) vs. 44016 (Allocation-site abstraction)

Adopted by all mainstream points-to analyses.

How?
Merging Objects → Over-Partition

Blindly Merging Objects → Precision Loss
Merging Objects \(\rightarrow\) Over-Partition

Blindly Merging Objects \(\rightarrow\) Precision Loss

\[ O_3 \xrightarrow{f} O_1 \]
\[ O_4 \xrightarrow{f} O_2 \]

inconsistent types
Merging Objects → Over-Partition

Blindly Merging Objects → Precision Loss

inconsistent types
Type-Consistent Objects

• Definition

\( O_i^T \) and \( O_j^T \) are type-consistent objects, if for every sequence of field names,

\[ f = f_1 \cdot f_2 \cdots \cdot f_n : \]

\( O_i^T.f \) and \( O_j^T.f \) point to the objects of the same types.
Type-Consistent Objects

• Definition

\( O_i^T \) and \( O_j^T \) are type-consistent objects, if for every sequence of field names,

\[ \overline{f} = f_1 \cdot f_2 \cdot \ldots \cdot f_n \]

\( O_i^T.\overline{f} \) and \( O_j^T.\overline{f} \) point to the objects of the same types.

MAHJONG only merges type-consistent objects
Type-Consistent Objects

- Example

Diagram: A graph with nodes labeled $O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8, O_9, O_{11}$ and edges labeled $f, g, h, k$. The nodes are connected in a specific pattern, indicating the type-consistent relationships among the objects.
Type-Consistent Objects

- Example

```
<table>
<thead>
<tr>
<th></th>
<th>O_1^T</th>
<th>O_2^T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>.f.h</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>.g</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.g.k</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
```
Type-Consistent Objects

- Example

<table>
<thead>
<tr>
<th></th>
<th>$O_1^T$</th>
<th>$O_2^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.f</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>.f.h</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>.g</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.g.k</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

$O_1^T$ and $O_2^T$ are type-consistent objects
How to Check Type-Consistency?
Our Solution: Sequential Automata

Check
Type-Consistency
of Objects

Test
Equivalence
of Automata
Sequential Automata

- 6-tuple \((Q, \Sigma, \delta, q_0, \Gamma, \gamma)\), where:
  - \(Q\) is a set of states
  - \(\Sigma\) is a set of input symbols
  - \(\delta\) is the next-state map: \(Q \times \Sigma \rightarrow \mathcal{P}(Q)\)
  - \(q_0\) is the initial state
  - \(\Gamma\) is a set of output symbols
  - \(\gamma\) is the output map: \(Q \rightarrow \Gamma\)
Check Type-Consistency of Objects

Test Equivalence of Automata

How?
<table>
<thead>
<tr>
<th>Objects</th>
<th>Automata</th>
</tr>
</thead>
<tbody>
<tr>
<td>A set of <strong>objects</strong></td>
<td><strong>Q</strong>: a set of <strong>states</strong></td>
</tr>
<tr>
<td>A set of <strong>field names</strong></td>
<td><strong>Σ</strong>: a set of <strong>input symbols</strong></td>
</tr>
<tr>
<td>The <strong>field points-to</strong> map</td>
<td><strong>δ</strong>: the <strong>next-state</strong> map</td>
</tr>
<tr>
<td>The <strong>object</strong> to be checked</td>
<td><strong>q₀</strong>: the <strong>initial state</strong></td>
</tr>
<tr>
<td>A set of <strong>types</strong></td>
<td><strong>Γ</strong>: a set of <strong>output symbols</strong></td>
</tr>
<tr>
<td>The <strong>object-to-type</strong> map</td>
<td><strong>γ</strong>: the <strong>output map</strong></td>
</tr>
</tbody>
</table>
Objects
- A set of **objects**
- A set of **field names**
- The field points-to map
- The object to be checked
- A set of **types**
- The object-to-type map

Automata
- **Q**: a set of **states**
- **Σ**: a set of **input symbols**
- **δ**: the next-state map
- **q₀**: the initial state
- **Γ**: a set of **output symbols**
- **γ**: the output map

objects ↔ states

\[ O_2^T, O_4^U, O_6^X, O_8^Y \]
Objects
- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata
- $Q$: a set of states
- $\Sigma$: a set of input symbols
- $\delta$: the next-state map
- $q_0$: the initial state
- $\Gamma$: a set of output symbols
- $\gamma$: the output map

field names $\leftrightarrow$ input symbols

$f, g, h, k$
Objects

- A set of objects
- A set of field names
- The **field points-to** map
- The object to be checked
- A set of types
- The **object-to-type** map

### Automata

- \( Q \): a set of states
- \( \Sigma \): a set of input symbols
- \( \delta \): the **next-state** map
- \( q_0 \): the initial state
- \( \Gamma \): a set of output symbols
- \( \gamma \): the output map

---

**field points-to map ↔ next-state map**

<table>
<thead>
<tr>
<th>( O_2^T )</th>
<th>f</th>
<th>( O_4^U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_2^T )</td>
<td>g</td>
<td>( O_6^X )</td>
</tr>
<tr>
<td>( O_4^U )</td>
<td>h</td>
<td>( O_8^Y )</td>
</tr>
<tr>
<td>( O_6^X )</td>
<td>k</td>
<td>( O_8^Y )</td>
</tr>
</tbody>
</table>
### Objects
- A set of **objects**
- A set of **field names**
- The **field points-to map**
- **The object to be checked**
- A set of **types**
- The **object-to-type map**

### Automata
- $Q$: a set of **states**
- $\Sigma$: a set of **input symbols**
- $\delta$: the **next-state map**
- $q_0$: the **initial state**
- $\Gamma$: a set of **output symbols**
- $\gamma$: the **output map**

---

checked object $\leftrightarrow$ initial state

$$O_2^T$$

---

[Diagram of an automaton with states $O_2$, $O_4$, $O_6$, $O_8$, $U$, $X$, $Y$, and transitions $f$, $h$, $g$, $k$.]
Objects
- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Types ↔ Output symbols

T, U, X, Y

Automata
- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- q₀: the initial state
- Γ: a set of output symbols
- γ: the output map
Objects

- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata

- $Q$: a set of states
- $\Sigma$: a set of input symbols
- $\delta$: the next-state map
- $q_0$: the initial state
- $\Gamma$: a set of output symbols
- $\gamma$: the output map

object-to-type map $\leftrightarrow$ output map

<table>
<thead>
<tr>
<th>$O_2^T$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_4^U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$O_6^X$</td>
<td>$X$</td>
</tr>
<tr>
<td>$O_8^Y$</td>
<td>$Y$</td>
</tr>
</tbody>
</table>
### Objects
- A set of **objects**
- A set of **field names**
- The **field points-to map**
- The **object** to be checked
- A set of **types**
- The **object-to-type map**

### Automata
- Q: a set of **states**
- Σ: a set of **input symbols**
- δ: the **next-state map**
- q₀: the **initial state**
- Γ: a set of **output symbols**
- γ: the **output map**

---

**Check Type-Consistency of Objects**

**Test Equivalence of Automata**
Test Equivalence of Automata

- Hopcroft-Karp algorithm*
  - Almost linear in terms of $|Q_{\text{larger}}|$
  - $Q_{\text{larger}}$: set of states of the larger automaton

Methodology
(MAHJONG)
Overview

Pre-Analysis

Field Points-to Graph (FPG)

fast but imprecise
e.g., context-insensitive

Points-to Analysis

precise but expensive
e.g., 3-object-sensitive

MAHJONG

NFA Builder

NFA<sub>O<sub>i</sub>T</sub>

∀<sub>O<sub>i</sub>T, O<sub>j</sub>T</sub>
in FPG

DFA<sub>O<sub>i</sub>T</sub>

DFA Converter

DFA<sub>O<sub>j</sub>T</sub>

Heap Modeler

Heap Abstraction

Automata Equivalence Checker

∀<sub>O<sub>i</sub>T, O<sub>j</sub>T</sub>

DFA<sub>O<sub>i</sub>T</sub> ≡ DFA<sub>O<sub>j</sub>T</sub>?
Working with Points-to Analysis

Original
- Allocation-site heap abstraction

New
- MAHJONG heap abstraction
  
  type-consistent objects
Implementation

- **1500** LOC of Java in total
- Integrated with **Doop**

- Can also be easily integrated to other points-to analysis frameworks
Evaluation
Evaluation - Research Questions

• **RQ1:** MAHJONG’s effectiveness as a pre-analysis
• **RQ2:** MAHJONG-based points-to-analysis
RQ1: MAHJONG’s Effectiveness as A Pre-Analysis

- Efficiency
  - Is MAHJONG lightweight for large programs?

- Heap partitioning
  - Can MAHJONG avoid heap over-partition?
## Pre-Analysis: Efficiency

<table>
<thead>
<tr>
<th></th>
<th>antlr</th>
<th>fop</th>
<th>luindex</th>
<th>pmd</th>
<th>chart</th>
<th>checkstyle</th>
<th>xalan</th>
<th>bloat</th>
<th>lusearch</th>
<th>JPC</th>
<th>findbugs</th>
<th>eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CI</strong></td>
<td>44.1</td>
<td>34.7</td>
<td>26.2</td>
<td>44.8</td>
<td>37.7</td>
<td>89.6</td>
<td>66.6</td>
<td>38.7</td>
<td>41.4</td>
<td>58.9</td>
<td>90.6</td>
<td>174.1</td>
</tr>
<tr>
<td><strong>FPG</strong></td>
<td>1.3</td>
<td>0.7</td>
<td>0.8</td>
<td>1.4</td>
<td>2.4</td>
<td>2.3</td>
<td>3.0</td>
<td>1.2</td>
<td>0.8</td>
<td>2.1</td>
<td>4.6</td>
<td>15.5</td>
</tr>
<tr>
<td><strong>MAHJONG</strong></td>
<td>1.3</td>
<td>1.1</td>
<td>1.1</td>
<td>1.5</td>
<td>1.9</td>
<td>4.0</td>
<td>3.1</td>
<td>1.7</td>
<td>1.0</td>
<td>4.5</td>
<td>3.2</td>
<td>21.4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>46.7</td>
<td>36.5</td>
<td>28.1</td>
<td>47.7</td>
<td>42.0</td>
<td>95.9</td>
<td>72.7</td>
<td>41.6</td>
<td>43.2</td>
<td>65.5</td>
<td>98.4</td>
<td>211.0</td>
</tr>
</tbody>
</table>

**CI**: Context-Insensitive points-to analysis  
**FPG**: Read Field Points-to Graph  
**MAHJONG**: Check automata equivalence, build heap abstraction

In total: 1 minute

**MAHJONG itself**: 3.8 seconds

Each program (on average)
Pre-Analysis: Heap Partition

Number of abstract objects created by the allocation-site abstraction and MAHJONG

Average reduction: 62%
RQ2: MAHJONG-Based Points-to Analysis

- Efficiency
  - Can MAHJONG accelerate points-to analysis?

- Precision
  - Can MAHJONG preserve precision for type-dependent clients?
Evaluated Points-to Analyses

- 5 mainstream context-sensitive points-to analyses:
  1. 2-call-site-sensitive analysis
  2. 2-type-sensitive analysis
  3. 3-type-sensitive analysis
  4. 2-object-sensitive analysis
  5. 3-object-sensitive analysis

- Time budget: 5 hours
Evaluated Clients

- Call graph construction
- Devirtualization
- May-fail casting
MAHJONG-Base Points-to Analysis: Results

- **Efficiency**
  Most precise (3-object-sensitive)
  *Speedup: 131X*

- **Precision**
  Call graph: -0.02%
  Devirtualization: -0.29%
  May-fail casting: -0%


MAHJONG-Base Points-to Analysis: Results

• Efficiency
  
  Most precise
  (3-object-sensitive)
  **Speedup: 131X**

  On average
  **Speedup: 15X**

• Precision
  
  Call graph: -0.02%
  Devirtualization: -0.29%
  May-fail casting: 0%

  Call graph: -0.02%
  Devirtualization: -0.18%
  May-fail casting: -0.03%
MAHJONG-Base Points-to Analysis: Results

- **Efficiency**
  - Most precise (3-object-sensitive)
  - *Speedup: 131X*
  - On average
  - *Speedup: 15X*

- **Precision**
  - Call graph: -0.02%
  - Devirtualization: -0.29%
  - May-fail casting: -0%

  - Call graph: -0.02%
  - Devirtualization: -0.18%
  - May-fail casting: -0.03%

For **checkstyle, xalan, lusearch, JPC, findbugs**

3-object-sensitive analysis:
- **without** MAHJONG, unscalable (> 5 hours)
- **with** MAHJONG, finish in 1 min ~ 84 mins (33 minutes on average)
Conclusion

• **MAHJONG**
  ◦ **Improve** significantly the **efficiency** of different point-to analyses
    • Call-site-, object- and type-sensitivity
  ◦ **Preserve** almost the same **precision** for type-dependent clients

• **Direct impact**
  ◦ Benefit many program analyses where **call graphs** are required
Thank you!