Static Program Analysis

Pointer Analysis

Nanjing University

Tian Tan

2020
Contents

1. Motivation
2. Introduction to Pointer Analysis
3. Key Factors of Pointer Analysis
4. Concerned Statements
Contents

1. Motivation
2. Introduction to Pointer Analysis
3. Key Factors of Pointer Analysis
4. Concerned Statements
Problem of CHA

```java
void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}
class One implements Number {
    public int get() { return 1; }
}
class Two implements Number {
    public int get() { return 2; }
}
```
Problem of CHA

```java
void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}
class Zero implements Number {
    public int get() { return 0; }
}
class One implements Number {
    public int get() { return 1; }
}
class Two implements Number {
    public int get() { return 2; }
}
```

CHA:

- call targets

Tian Tan @ Nanjing University
Problem of CHA

```java
void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}

class One implements Number {
    public int get() { return 1; }
}

class Two implements Number {
    public int get() { return 2; }
}
```

CHA: based on class hierarchy
- 3 call targets

Tian Tan @ Nanjing University
Problem of CHA

void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}

class One implements Number {
    public int get() { return 1; }
}

class Two implements Number {
    public int get() { return 2; }
}

CHA: based on class hierarchy
• 3 call targets

Constant propagation
• $x = ?$
Problem of CHA

CHAs: based on
- class hierarchy
  - 3 call targets

Constant propagation
- \( x = \text{NAC} \)

```java
void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}
class One implements Number {
    public int get() { return 1; }
}
class Two implements Number {
    public int get() { return 2; }
}
```

Tian Tan @ Nanjing University
Problem of CHA

void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}

class One implements Number {
    public int get() { return 1; }
}

class Two implements Number {
    public int get() { return 2; }
}

CHA: based on only considers class hierarchy
- 3 call targets
- 2 false positives

Constant propagation
- x = NAC imprecise
Via Pointer Analysis

void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}

class One implements Number {
    public int get() { return 1; }
}

class Two implements Number {
    public int get() { return 2; }
}

CHA: based on only considers class hierarchy
• 3 call targets
• 2 false positives

Constant propagation
• \( x = \text{NAC} \) imprecise

Pointer analysis: based on points-to relation
• 1 call target

Tian Tan @ Nanjing University
Via Pointer Analysis

void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}
class Zero implements Number {
    public int get() { return 0; }
}
class One implements Number {
    public int get() { return 1; }
}
class Two implements Number {
    public int get() { return 2; }
}

CHA: based on only considers class hierarchy
• 3 call targets
• 2 false positives

Constant propagation
• x = NAC

Pointer analysis: based on points-to relation
• 1 call target

Constant propagation
• x = 1
Via Pointer Analysis

void foo() {
    Number n = new One();
    int x = n.get();
}

interface Number {
    int get();
}

class Zero implements Number {
    public int get() { return 0; }
}

class One implements Number {
    public int get() { return 1; }
}

class Two implements Number {
    public int get() { return 2; }
}

CHA: based on only considers class hierarchy
• 3 call targets
• 2 false positives

Constant propagation
• x = NAC imprecise

Pointer analysis: based on points-to relation
• 1 call target
• 0 false positive

Constant propagation
• x = 1 precise
Contents

1. Motivation
2. Introduction to Pointer Analysis
3. Key Factors of Pointer Analysis
4. Concerned Statements
Pointer Analysis

• A fundamental static analysis
  • Computes which memory locations a pointer can point to
Pointer Analysis

• A fundamental static analysis
  • Computes which memory locations a pointer can point to

• For object-oriented programs (focus on Java)
  • Computes which objects a pointer (variable or field) can point to
Pointer Analysis

• A fundamental static analysis
  • Computes which memory locations a pointer can point to

• For object-oriented programs (focus on Java)
  • Computes which objects a pointer (variable or field) can point to

• Regarded as a may-analysis
  • Computes an over-approximation of the set of objects that a pointer can point to, i.e., we ask “a pointer may point to which objects?”
Pointer Analysis

• A fundamental static analysis
  • Computes which memory locations a pointer can point to

• For object-oriented programs (focus on Java)
  • Computes which objects a pointer (variable or field) can point to

• Regarded as a may-analysis
  • Computes an over-approximation of the set of objects that a pointer can point to, i.e., we ask “a pointer may point to which objects?”

A research area with 40+ years of history


Still an active area today

➢ OOPSLA’18, FSE’18, TOPLAS’19, OOPSLA’19, TOPLAS’20, …
Example

“Which objects a pointer can point to?”

void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
Example

“Which objects a pointer can point to?”

Program

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

Points-to relations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
</tbody>
</table>
Example

“Which objects a pointer can point to?”

Program

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

Points-to relations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>?</td>
</tr>
</tbody>
</table>
Example

“Which objects a pointer can point to?”

Program

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

Points-to relations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>new A</td>
</tr>
<tr>
<td>b</td>
<td>new B</td>
</tr>
</tbody>
</table>
Example

“Which objects a pointer can point to?”

Program

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

Points-to relations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>new A</td>
</tr>
<tr>
<td>b</td>
<td>new B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>new A.b</td>
<td>new B</td>
</tr>
</tbody>
</table>
Example

“Which objects a pointer can point to?”

Program

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

Points-to relations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>new A</td>
</tr>
<tr>
<td>b</td>
<td>new B</td>
</tr>
<tr>
<td>y</td>
<td>?</td>
</tr>
</tbody>
</table>

Field Object

<table>
<thead>
<tr>
<th>Field</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>new A</td>
<td>b</td>
</tr>
</tbody>
</table>
Example

“Which **objects** a pointer can point to?”

```java
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>new A</td>
</tr>
<tr>
<td>b</td>
<td>new B</td>
</tr>
<tr>
<td>y</td>
<td>new B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>new A.b</td>
<td>new B</td>
</tr>
</tbody>
</table>
Example

“Which objects a pointer can point to?”

```
void foo() {
    A a = new A();
    B x = new B();
    a.setB(x);
    B y = a.getB();
}

class A {
    B b;
    void setB(B b) { this.b = b; }
    B getB() { return this.b; }
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A</td>
</tr>
<tr>
<td>x</td>
<td>new B</td>
</tr>
<tr>
<td>this</td>
<td>new A</td>
</tr>
<tr>
<td>b</td>
<td>new B</td>
</tr>
<tr>
<td>y</td>
<td>new B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>new A.b</td>
<td>new B</td>
</tr>
</tbody>
</table>
Pointer Analysis and Alias Analysis

Two closely related but different concepts

- Pointer analysis: which objects a pointer can point to?
- Alias analysis: can two pointers point to the same object?
Pointer Analysis and Alias Analysis

Two closely related but different concepts

• Pointer analysis: which objects a pointer can point to?
• Alias analysis: can two pointers point to the same object?

If two pointers, say p and q, refer to the same object, then p and q are aliases

```java
p = new C();
q = p;
x = new X();
y = new Y();
```

p and q are aliases
x and y are not aliases
Pointer Analysis and Alias Analysis

Two closely related but different concepts

• Pointer analysis: which objects a pointer can point to?
• Alias analysis: can two pointers point to the same object?

If two pointers, say \( p \) and \( q \), refer to the same object, then \( p \) and \( q \) are aliases

\[
p = \text{new } C(); \\
q = p; \\
x = \text{new } X(); \\
y = \text{new } Y();
\]

\( p \) and \( q \) are aliases
\( x \) and \( y \) are not aliases

Alias information can be derived from points-to relations
Applications of Pointer Analysis

• Fundamental information
  o Call graph, aliases, ...

• Compiler optimization
  o Virtual call inlining, ...

• Bug detection
  o Null pointer detection, ...

• Security analysis
  o Information flow analysis,

• And many more ...

“Pointer analysis is one of the most fundamental static program analyses, on which virtually all others are built.”*
Applications of Pointer Analysis

• Fundamental information
  o Call graph, aliases, ...
• Compiler optimization
  o Virtual call inlining, ...
• Bug detection
  o Null pointer detection, ...
• Security analysis
  o Information flow analysis,
• And many more ...

“Pointer analysis is one of the most fundamental static program analyses, on which virtually all others are built.”*

Contents

1. Motivation
2. Introduction to Pointer Analysis
3. Key Factors of Pointer Analysis
4. Concerned Statements
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the precision and efficiency of the system
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the precision and efficiency of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the **precision** and **efficiency** of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the **precision** and **efficiency** of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
</table>
| Heap abstraction        | How to model heap memory?                                              | • Allocation-site
                          |                                          | • Storeless                           |
| Context sensitivity     | How to model calling contexts?                                         | • Context-sensitive
                          |                                          | • Context-insensitive                  |
| Flow sensitivity        | How to model control flow?                                             | • Flow-sensitive
                          |                                          | • Flow-insensitive                     |
| Analysis scope          | Which parts of program should be analyzed?                             | • Whole-program
                          |                                          | • Demand-driven                        |
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the **precision** and **efficiency** of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
</table>
| Heap abstraction     | How to model heap memory?                                | • Allocation-site  
|                      |                                                           | • Storeless                                 |
| Context sensitivity  | How to model calling contexts?                           | • Context-sensitive  
|                      |                                                           | • Context-insensitive                       |
| Flow sensitivity     | How to model control flow?                               | • Flow-sensitive  
|                      |                                                           | • Flow-insensitive                          |
| Analysis scope       | Which parts of program should be analyzed?               | • Whole-program  
|                      |                                                           | • Demand-driven                            |
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the precision and efficiency of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>
Heap Abstraction

How to model heap memory?

- In dynamic execution, the number of heap objects can be unbounded due to loops and recursion

```java
for (...) {
    A a = new A();
}
```
Heap Abstraction

How to model heap memory?

• In dynamic execution, the number of heap objects can be unbounded due to loops and recursion

```java
for (...) {
    A a = new A();
}
```

• To ensure termination, heap abstraction models dynamically allocated, unbounded concrete objects as **finite abstract objects** for static analysis
Heap Abstraction

How to model heap memory?

• In dynamic execution, the number of heap objects can be unbounded due to loops and recursion

```java
for (...) {
    A a = new A();
}
```

• To ensure termination, heap abstraction models dynamically allocated, unbounded concrete objects as finite abstract objects for static analysis

![Diagram showing dynamic execution and static analysis with abstracted bounded and unbounded objects]
Heap Abstraction

Figure 2. Heap memory can be modeled as storeless, store based, or hybrid. These models are summarized using allocation sites, $k$-limiting, patterns, variables, other generic instrumentation predicates, or higher-order logics.

Heap Abstraction

Figure 2. Heap memory can be modeled as storeless, store based, or hybrid. These models are summarized using allocation sites, $k$-limiting, patterns, variables, other generic instrumentation predicates, or higher-order logics.

Vini Kanvar, Uday P. Khedker, "Heap Abstractions for Static Analysis". ACM CSUR 2016
Allocation-Site Abstraction

The most commonly-used heap abstraction

• Model concrete objects by their allocation sites
• One abstract object per allocation site to represent all its allocated concrete objects
Allocation-Site Abstraction

The most commonly-used heap abstraction

• Model concrete objects by their allocation sites
• One abstract object per allocation site to represent all its allocated concrete objects

1 for (i = 0; i < 3; ++i) {
2     a = new A();
3     ...  
4 }
Allocation-Site Abstraction

The most commonly-used heap abstraction

• Model concrete objects by their allocation sites
• One abstract object per allocation site to represent all its allocated concrete objects

```
1 for (i = 0; i < 3; ++i) {
2     a = new A();
3     ...
4 }
```

Dynamic execution

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Abstracted Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$o_2$, iteration 0</td>
</tr>
<tr>
<td>1</td>
<td>$o_2$, iteration 1</td>
</tr>
<tr>
<td>2</td>
<td>$o_2$, iteration 2</td>
</tr>
</tbody>
</table>

Allocation-site abstraction
Allocation-Site Abstraction

The most commonly-used heap abstraction

- Model concrete objects by their allocation sites
- One abstract object per allocation site to represent all its allocated concrete objects

```
1 for (i = 0; i < 3; ++i) {
2    a = new A();
3    ...
4 }
```

The number of allocation sites in a program is bounded, thus the abstract objects must be finite.

Tian Tan @ Nanjing University
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the **precision** and **efficiency** of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
## Context Sensitivity

### How to model calling contexts?

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinguish different calling contexts of a method</td>
<td>Merge all calling contexts of a method</td>
</tr>
<tr>
<td>Analyze each method multiple times, once for each context</td>
<td>Analyze each method once</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
# Context Sensitivity

## How to model calling contexts?

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Distinguish</strong> different calling contexts of a method</td>
<td>Merge all calling contexts of a method</td>
</tr>
<tr>
<td>Analyze each method multiple times, once for each context</td>
<td>Analyze each method once</td>
</tr>
</tbody>
</table>

```cpp
Context 1:
void foo(T p) {
    ...
}
```

```cpp
Context 2:
void foo(T p) {
    ...
}
```

Tian Tan @ Nanjing University
Context Sensitivity

How to model calling contexts?

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinguish different calling contexts of a method</td>
<td>Merge all calling contexts of a method</td>
</tr>
<tr>
<td>Analyze each method multiple times, once for each context</td>
<td>Analyze each method once</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
## Context Sensitivity

### How to model calling contexts?

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinguish different calling contexts of a method</td>
<td>Merge all calling contexts of a method</td>
</tr>
<tr>
<td>Analyze each method multiple times, once for each context</td>
<td>Analyze each method once</td>
</tr>
</tbody>
</table>

---

Tian Tan @ Nanjing University

---

### Context 1:
```java
void foo(T p) {
    ...
}
```

### Context 2:
```java
void foo(T p) {
    ...
}
```

---

### Merge data flow, may lose precision

---

52
## Context Sensitivity

### How to model calling contexts?

<table>
<thead>
<tr>
<th>Context-sensitive</th>
<th>Context-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinguish different calling contexts of a method</td>
<td>Merge all calling contexts of a method</td>
</tr>
<tr>
<td>Analyze each method multiple times, once for each context</td>
<td>Analyze each method once</td>
</tr>
</tbody>
</table>

Very useful technique
Significantly improve precision
More details in later lectures

We start with **this**

```
Context 1: void foo(T p) {
    ...
}
```

```
Context 2: void foo(T p) {
    ...
}
```

```
void foo(T p) {
    ...
}
```
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the precision and efficiency of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

So far, all data-flow analyses we have learnt are **flow-sensitive**
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```
1 c = new C();
2 c.f = "x";
3 s = c.f;
4 c.f = "y";
```
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```java
1 c = new C();
2 c.f = "x";
3 s = c.f;
4 c.f = "y";
```
## Flow Sensitivity

### How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```java
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

Tian Tan @ Nanjing University
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```plaintext
1  c = new C();
2  c.f = "x";
3  s = c.f;
4  c.f = "y";
```

Tian Tan @ Nanjing University
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

1. \( c = \text{new } C() \);
2. \( c.f = "x" \);
3. \( s = c.f \);
4. \( c.f = "y" \);

Tian Tan @ Nanjing University
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

1. \( c = \text{new} \ C(); \)
2. \( c.f = "x"; \)
3. \( s = c.f; \)
4. \( c.f = "y"; \)
# Flow Sensitivity

## How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```plaintext
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

**Diagram:**

- `c` maps to `{o1}`
- `o1.f` maps to `{"x"}`
- `s` maps to `{"x"}`
- `o1.f` maps to `{"y"}`

Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

1. c = new C();
2. c.f = "x";
3. s = c.f;
4. c.f = "y";

---

```
c ➝ {o1}
```

```c
1 c = new C();
2 c.f = "x";
3 s = c.f;
4 c.f = "y";
```
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```java
1. c = new C();
2. c.f = "x";
3. s = c.f;
4. c.f = "y";
```

```
c ➔ {o1}
```

```
c ➔ {o1}
  o1.f ➔ {"x"}
```

```
c ➔ {o1}
  o1.f ➔ {"x"}
```

```
c ➔ {o1}
  o1.f ➔ {"x"}
  s ➔ {"x"}
```

```
c ➔ {o1}  
  o1.f ➔ {"y"}
```

```
c ➔ {o1}  
  o1.f ➔ {"y"}
  s ➔ {"x"}
```
## Flow Sensitivity

### How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

- Flow-sensitive examples:
  - \( c \rightarrow \{ o_1 \} \)
  - \( c \rightarrow \{ o_1 \} \)
  - \( o_1.f \rightarrow \{ "x" \} \)

- Flow-insensitive examples:
  - \( c \rightarrow \{ o_1 \} \)
  - \( o_1.f \rightarrow \{ "x", "y" \} \)

```java
1   c = new C();
2   c.f = "x";
3   s = c.f;
4   c.f = "y";
```

The execution flow and the points-to relations are shown visually with a diagram.
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

```
c → {o1}
o1.f → {"x"}
c → {o1}
o1.f → {"x"}
s → {"x"}
```

```
c → {o1}
o1.f → {"x", "y"}
s → ?
```
# Flow Sensitivity

## How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

### Code Examples

1. `c = new C();`
2. `c.f = "x";`
3. `s = c.f;`
4. `c.f = "y";`

- **Flow-sensitive**
  - `c ➝ \{o1\}`
  - `c ➝ \{o1\}`
  - `o1.f ➝ \{"x"\}`
  - `s ➝ \{"x"\}`

- **Flow-insensitive**
  - `c ➝ \{o1\}`
  - `o1.f ➝ \{"x", "y"\}`
  - `s ➝ \{"x", "y"\}`

- **Flow-sensitive** (updated)
  - `c ➝ \{o1\}`
  - `o1.f ➝ \{"x"\}`
  - `s ➝ \{"x"\}`

- **Flow-insensitive** (updated)
  - `c ➝ \{o1\}`
  - `o1.f ➝ \{"y"\}`
  - `s ➝ \{"x"\}`
Flow Sensitivity

How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

1. `c = new C();`
2. `c.f = "x";`
3. `s = c.f;`
4. `c.f = "y";`

- `c` ➔ `{o1}`
- `o1.f` ➔ `{"x"}`
- `s` ➔ `{"x"}`

- `c` ➔ `{o1}`
- `o1.f` ➔ `{"y"}`
- `s` ➔ `{"x"}`

False positive
# Flow Sensitivity

## How to model control flow?

<table>
<thead>
<tr>
<th>Flow-sensitive</th>
<th>Flow-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Respect the execution order of the statements</td>
<td>Ignore the control-flow order, treat the program as a set of unordered statements</td>
</tr>
<tr>
<td>Maintain a map of points-to relations at each program location</td>
<td>Maintain one map of points-to relations for the whole program</td>
</tr>
</tbody>
</table>

```plaintext
1. c = new C();
2. c.f = "x";
3. s = c.f;
4. c.f = "y";
```

Chosen in this course
Key Factors in Pointer Analysis

- Pointer analysis is a complex system
- Multiple factors affect the **precision** and **efficiency** of the system

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
# Analysis Scope

**Which parts of program should be analyzed?**

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect specific sites of interest (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>
Analysis Scope

Which parts of program should be analyzed?

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect specific sites of interest (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

```
1 x = new A();
2 y = x;
3 ...
4 z = new T();
5 z.bar();
```
### Analysis Scope

**Which parts of program should be analyzed?**

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect specific sites of interest (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

```plaintext
1  x = new A();
2  y = x;
3  ...
4  z = new T();
5  z.bar();
```
## Analysis Scope

**Which parts of program should be analyzed?**

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect <strong>specific sites of interest</strong> (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

1. \( x = \text{new } A(); \)
2. \( y = x; \)
3. ...\n4. \( z = \text{new } T(); \)
5. \( z.\text{bar}(); \)

What points-to information do we need?

**Client:** call graph construction

**Site of interest:** line 5
Analysis Scope

Which parts of program should be analyzed?

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect specific sites of interest (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

\[
x \rightarrow \{01\} \\
y \rightarrow \{01\} \\
z \rightarrow \{04\}
\]

1. \(x = \text{new } A();\)
2. \(y = x;\)
3. \(...\)
4. \(z = \text{new } T();\)
5. \(z.\text{bar}();\)

\[
z \rightarrow \{04\}
\]

Client: call graph construction
Site of interest: line 5
Analysis Scope

Which parts of program should be analyzed?

<table>
<thead>
<tr>
<th>Whole-program</th>
<th>Demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute points-to information for all pointers in the program</td>
<td>Only compute points-to information for the pointers that may affect specific sites of interest (on demand)</td>
</tr>
<tr>
<td>Provide information for all possible clients</td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

Chosen in this course

```
x = new A();
y = x;
...
z = new T();
z.bar();
```

Client: call graph construction
Site of interest: line 5
# Pointer Analysis in This Course

<table>
<thead>
<tr>
<th>Factor</th>
<th>Problem</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap abstraction</td>
<td>How to model heap memory?</td>
<td>• Allocation-site</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Storeless</td>
</tr>
<tr>
<td>Context sensitivity</td>
<td>How to model calling contexts?</td>
<td>• Context-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Context-insensitive</td>
</tr>
<tr>
<td>Flow sensitivity</td>
<td>How to model control flow?</td>
<td>• Flow-sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flow-insensitive</td>
</tr>
<tr>
<td>Analysis scope</td>
<td>Which parts of program should be analyzed?</td>
<td>• Whole-program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Demand-driven</td>
</tr>
</tbody>
</table>
Contents

1. Motivation
2. Introduction to Pointer Analysis
3. Key Factors of Pointer Analysis
4. Concerned Statements
What Do We Analyze?

• Modern languages typically have many kinds of statements
  • if-else
  • switch-case
  • for/while/do-while
  • break/continue
  • ...
What Do We Analyze?

• Modern languages typically have many kinds of statements
  • if-else
  • switch-case
  • for/while/do-while
  • break/continue
  • ...

• We only focus on pointer-affecting statements

Do not directly affect pointers
Ignored in pointer analysis
Pointers in Java

• Local variable: x

• Static field: C.f

• Instance field: x.f

• Array element: array[i]
Pointers in Java

• Local variable: \texttt{x}

• Static field: \texttt{C.f}

• Instance field: \texttt{x.f}

• Array element: \texttt{array[i]}
Pointers in Java

• Local variable: x

• Static field: C.f

Sometimes referred as global variable

• Instance field: x.f

• Array element: array[i]
Pointers in Java

• Local variable: x
• Static field: C.f
• Instance field: x.f
• Array element: array[i]

Modeled as an object (pointed by x) with a field f
Points in Java

• Local variable: x

• Static field: C.f

• Instance field: x.f

• Array element: array[i]

```
array = new String[10];
array[0] = "x";
array[1] = "y";
s = array[0];
```

```
array = new String[];
array.arr = "x";
array.arr = "y";
s = array.arr;
```

Ignore indexes. Modeled as an object (pointed by array) with a single field, say arr, which may point to any value stored in array

Real code

Perspective of pointer analysis
Pointers in Java

• Local variable: x

• Static field: C.f

• Instance field: x.f

• Array element: array[i]
Pointer-Affecting Statements

New \hspace{1cm} x = \texttt{new} T()

Assign \hspace{1cm} x = y

Store \hspace{1cm} x.f = y

Load \hspace{1cm} y = x.f

Call \hspace{1cm} r = x.k(a, \ldots)
Pointer-Affecting Statements

New
\[ x = \text{new } T() \]

Assign
\[ x = y \]

Store
\[ x.f = y \]

Load
\[ y = x.f \]

Call
\[ r = x.k(a, \ldots) \]

Complex memory-accesses will be converted to three-address code by introducing temporary variables

\[ x.f.g.h = y; \]
\[ t1 = x.f \]
\[ t2 = t1.g \]
\[ t2.h = y; \]
Pointer-Affecting Statements

- **New**  
  \[ x = \text{new} \ T() \]

- **Assign**  
  \[ x = y \]

- **Store**  
  \[ x.f = y \]

- **Load**  
  \[ y = x.f \]

- **Call**  
  \[ r = x.k(a, \ldots) \]

- **Static call**  
  \[ C.foo() \]

- **Special call**  
  \[ super.foo() / x.<init>() / this.privateFoo() \]

- **Virtual call**  
  \[ x.foo() \]
Pointer-Affecting Statements

New       x = new T()
Assign    x = y
Store     x.f = y
Load      y = x.f
Call      r = x.k(a, ...)

- Static call          C.foo()
- Special call         super.foo() / x.<init>() / this.privateFoo()
- Virtual call         x.foo()  focus
The You Need To Understand in This Lecture

• What is pointer analysis?
• Understand the key factors of pointer analysis
• Understand what we analyze in pointer analysis
计算机科学与技术系

程序语言与计算

静态与动态分析

组态与分析

程序与计算