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

Tian Tan @ Nanjing University
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
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

Constant propagation
- x = ?

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 = NAC
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

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

**interface Number {**
```java
    int get();
}
```

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

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

**class Two implements Number {**
```java
    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
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

Constant propagation
• \( x = 1 \)
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; }
}

Via Pointer Analysis

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

Tian Tan @ Nanjing University
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?”

```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; }
}
```
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>

```
Field           Object
---------------
new A.b          new 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>
<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>

<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; }
}
```

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>new B</td>
</tr>
</tbody>
</table>

Field | Object |
------|--------|
new A.b | new B  |
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

\[
p = \text{new } C(); \quad \text{p and q are aliases}
q = p;
\]

\[
x = \text{new } X(); \quad \text{x and y are not aliases}
y = \text{new } Y();
\]
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

\[
\begin{align*}
p &= \text{new } C(); \\
q &= p; \\
x &= \text{new } X(); \\
y &= \text{new } Y();
\end{align*}
\]

Alias information can be derived from points-to relations

Tian Tan @ Nanjing University
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>
</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>
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>
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
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
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.

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

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

Dynamic execution
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

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

Dynamic execution

 Allocation-site abstraction

\(o_2, \text{iteration } i = 0\)
\(o_2, \text{iteration } i = 1\)
\(o_2, \text{iteration } i = 2\)

\(o_2\)
Allocation-Site 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.

The most commonly-used heap abstraction

Dynamic execution

Allocation-site abstraction
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>
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>
## 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>

```plaintext
def foo(T p):
    ...

a.foo(x);
b.foo(y);
```

**Context 1:**
```plaintext
def foo(T p):
    ...

}  

**Context 2:**
```plaintext
def 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>

```java
a.foo(x); b.foo(y);

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?

### Context-sensitive

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

### Context-insensitive

Very useful technique

Significantly improve precision

More details in later lectures

---

We start with this

```java
void foo(T p) {
    ...
}

a.foo(x);

Context 1:

Context 2:

b.foo(y);
```
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>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>

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>

```
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>

### Example Code:

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

### Diagram:

- **Flow-sensitive**:
  - `c` points to `{o1}`
  - `o1.f` points to `{"x"}`

- **Flow-insensitive**:
  - `s` points to `?`
## 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>

**Example:**

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

<table>
<thead>
<tr>
<th>c $\rightarrow$</th>
<th>{o₁}</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁.f $\rightarrow$</td>
<td>{&quot;x&quot;}</td>
</tr>
<tr>
<td>s $\rightarrow$</td>
<td>{&quot;x&quot;}</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>

```
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>

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}

o1.f ➝ {"x"}

s ➝ {"x"}
```

```
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";
```

---

```
1 c ➝ \{o1\}
2 o1.f ➝ \{"x"\}
3 c ➝ \{o1\}
4 o1.f ➝ \{"x"\}
5 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. \( c = \text{new} \ C(); \)
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>

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

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

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

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

- `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>

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

```
c ➝ \{o1\}
oc1.f ➝ \{"x"\}
c ➝ \{o1\}
oc1.f ➝ \{"x", "y"\}
s ➝ \{"x", "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";

**False positive**

```plaintext
  c ➝ {o1}
  o1.f ➝ {"x"}
  s ➝ {"x"}
```

```plaintext
  c ➝ {o1}
  o1.f ➝ {"x", "y"}
  s ➝ {"x", "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>

### Example

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

### Chosen in this course

- `c` points to `o1`
- `o1.f` points to `"x"`
- `s` points to `"x"`
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>

```java
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>

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

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

Client: call graph construction
Site of interest: line 5

What points-to information do we need?
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>

Tian Tan @ Nanjing University

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

\textbf{Client}: call graph construction

\textbf{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 <strong>all possible clients</strong></td>
<td>Provide information for specific clients</td>
</tr>
</tbody>
</table>

### Chosen in this course

```
x  →  {o1}
y  →  {o1}
z  →  {o4}
```

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

**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: x
- Static field: C.f
- Instance field: x.f
- Array element: 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];

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

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

Real code

Perspective of pointer analysis

Tian Tan @ Nanjing University
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 = \text{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 \[ \text{C.foo()} \]
• Special call \[ \text{super.foo()}/\text{x.<init>())/this.privateFoo()} \]
• Virtual call \[ \text{x.foo()} \]
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**
  \[ C.\text{foo}() \]

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

- **Virtual call**
  \[ x.\text{foo}() \text{ focus} \]
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
Contents

1. Pointer Analysis: Rules
2. How to Implement Pointer Analysis
3. Pointer Analysis: Algorithms
4. Pointer Analysis with Method Calls
Contents

1. Pointer Analysis: Rules
2. How to Implement Pointer Analysis
3. Pointer Analysis: Algorithms
4. Pointer Analysis with Method Calls
Pointer-Affecting Statements

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

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

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

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

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

First focus on these statements (suppose the program has just one method)

Will come back to this in pointer analysis with method calls
Domain and Notations

Variables: \( x, y \in V \)

Fields: \( f, g \in F \)

Objects: \( o_i, o_j \in O \)

Instance fields: \( o_i.f, o_j.g \in O \times F \)

Pointers: \( \text{Pointer} = V \cup (O \times F) \)

Points-to relations: \( pt : \text{Pointer} \rightarrow \mathcal{P}(O) \)

\( \mathcal{P}(O) \) denotes the powerset of \( O \)
\( pt(p) \) denotes the points-to set of \( p \)
## Rules

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$i: x = \text{new } T()$</td>
<td>$o_i \in pt(x)$</td>
</tr>
<tr>
<td>Assign</td>
<td>$x = y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$o_i \in pt(y)$</td>
<td>$o_i \in pt(x)$</td>
</tr>
<tr>
<td>Store</td>
<td>$x.f = y$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$o_i \in pt(x)$, $o_j \in pt(y)$</td>
<td>$o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td>Load</td>
<td>$y = x.f$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$o_i \in pt(x)$, $o_j \in pt(o_i.f)$</td>
<td>$o_j \in pt(y)$</td>
</tr>
</tbody>
</table>
## Rules

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$i: x = \text{new } T()$</td>
<td>$o_i \in pt(x)$ ← unconditional</td>
</tr>
<tr>
<td>Assign</td>
<td>$x = y$</td>
<td>$o_i \in pt(y)$ ← premises $o_i \in pt(x)$ ← conclusion</td>
</tr>
<tr>
<td>Store</td>
<td>$x.f = y$</td>
<td>$o_i \in pt(x), o_j \in pt(y)$ $o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td>Load</td>
<td>$y = x.f$</td>
<td>$o_i \in pt(x), o_j \in pt(o_i.f)$ $o_j \in pt(y)$</td>
</tr>
</tbody>
</table>
Rule: New

\[
\frac{\text{i: } x = \text{new } T()}{o_i \in pt(x)}
\]

Conclusion
Rule: Assign

\[ o_i \in pt(y) \]
\[ \frac{o_i \in pt(y)}{o_i \in pt(x)} \]

Premises

Conclusion

\[ x = y \]
Rule: Store

\[
o_i \in pt(x), \ o_j \in pt(y) \\
\hline \\
o_j \in pt(o_i \cdot f)
\]

Premises

Conclusion

\[o_i \xrightarrow{f} o_j\]

\[x \cdot f = y\]
Rule: Load

\[ o_i \in pt(x), \quad o_j \in pt(o_i \cdot f) \]

\[ \frac{}{o_j \in pt(y)} \]

--- Premises

\[ y = x \cdot f \]

--- Conclusion
# Rules

<table>
<thead>
<tr>
<th>Kind</th>
<th>Rule</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$o_i \in pt(x)$</td>
<td>$i: x = \text{new } T()$</td>
</tr>
<tr>
<td>Assign</td>
<td>$o_i \in pt(y)$</td>
<td>$i \in pt(x)$</td>
</tr>
<tr>
<td>Store</td>
<td>$o_i \in pt(x), o_j \in pt(y)$</td>
<td>$o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td>Load</td>
<td>$o_i \in pt(x), o_j \in pt(o_i.f)$</td>
<td>$o_j \in pt(y)$</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University
Contents

1. Pointer Analysis: Rules
2. How to Implement Pointer Analysis
3. Pointer Analysis: Algorithms
4. Pointer Analysis with Method Calls
Our Pointer Analysis Algorithms

• A complete whole-program pointer analysis

• Carefully designed for understandability

• Easy to follow and implement
How to Implement Pointer Analysis?

- Essentially, pointer analysis is to **propagate** points-to information among pointers (variables & fields)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new } T() )</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y) ), ( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x) ), ( o_j \in pt(y) ), ( o_j \in pt(o_i.f) )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x) ), ( o_j \in pt(o_i.f) ), ( o_j \in pt(y) )</td>
</tr>
</tbody>
</table>
How to Implement Pointer Analysis?

• Essentially, pointer analysis is to propagate points-to information among pointers (variables & fields)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>i: x = new T()</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Assign</td>
<td>x = y</td>
<td>( o_i \in pt(y), o_i \in pt(x) )</td>
</tr>
<tr>
<td>Store</td>
<td>x.f = y</td>
<td>( o_i \in pt(x), o_j \in pt(y), o_j \in pt(o_i.f) )</td>
</tr>
<tr>
<td>Load</td>
<td>y = x.f</td>
<td>( o_i \in pt(x), o_j \in pt(o_i.f), o_j \in pt(y) )</td>
</tr>
</tbody>
</table>


Pointer analysis as solving a system of inclusion constraints for pointers

Referred as *Andersen-style analysis*
How to Implement Pointer Analysis?

• Essentially, pointer analysis is to **propagate** points-to information among pointers (variables & fields)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>i: x = new T()</td>
<td>$o_i \in pt(x)$</td>
</tr>
<tr>
<td>Assign</td>
<td>x = y</td>
<td>$o_i \in pt(y)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o_i \in pt(x)$</td>
</tr>
<tr>
<td>Store</td>
<td>x.f = y</td>
<td>$o_i \in pt(x)$, $o_j \in pt(y)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td>Load</td>
<td>y = x.f</td>
<td>$o_i \in pt(x)$, $o_j \in pt(o_i.f)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o_j \in pt(y)$</td>
</tr>
</tbody>
</table>

Key to implementation: when $pt(x)$ is **changed**, **propagate** the **changed part** to the **related pointers** of $x$
How to Implement Pointer Analysis?

• Essentially, pointer analysis is to **propagate** points-to information among pointers (variables & fields)

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new } T() )</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y) ) ( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x) ), ( o_j \in pt(y) ) ( o_j \in pt(o_i.f) )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x) ), ( o_j \in pt(o_i.f) ) ( o_j \in pt(y) )</td>
</tr>
</tbody>
</table>

**Solution**
- We use a **graph** to connect related pointers
- When \( pt(x) \) changes, propagate the changed part to \( x \)’s **successors**

Key to implementation: when \( pt(x) \) is **changed**, **propagate** the **changed part** to the **related pointers** of **\( x \)**
Pointer Flow Graph (PFG)

Pointer flow graph of a program is a directed graph that expresses how objects flow among the pointers in the program.
Pointer Flow Graph (PFG)

Pointer flow graph of a program is a directed graph that expresses how objects flow among the pointers in the program.

• Nodes: Pointer = V U (O × F)
  A node \( n \) represents a variable or a field of an abstract object

• Edges: Pointer × Pointer
  An edge \( x \rightarrow y \) means that the objects pointed by pointer \( x \) may flow to (and also be pointed to by) pointer \( y \)
Pointer Flow Graph: Edges

- PFG edges are added according to the statements of the program and the corresponding rules

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new} \ T() )</td>
<td>( o_i \in pt(x) )</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y), o_i \in pt(x) )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x), o_j \in pt(y) )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x), o_j \in pt(o_i.f) )</td>
</tr>
</tbody>
</table>

Tian Tan @ Nanjing University

112
### Pointer Flow Graph: Edges

- PFG edges are added according to the statements of the program and the corresponding rules

<table>
<thead>
<tr>
<th>Kind</th>
<th>Statement</th>
<th>Rule</th>
<th>PFG Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>( i: x = \text{new} \ T() )</td>
<td>( o_i \in pt(x) )</td>
<td>N/A</td>
</tr>
<tr>
<td>Assign</td>
<td>( x = y )</td>
<td>( o_i \in pt(y) ), ( o_i \in pt(x) )</td>
<td>( x \leftarrow y )</td>
</tr>
<tr>
<td>Store</td>
<td>( x.f = y )</td>
<td>( o_i \in pt(x), \ o_j \in pt(y) ), ( o_j \in pt(o_i.f) )</td>
<td>( o_i.f \leftarrow y )</td>
</tr>
<tr>
<td>Load</td>
<td>( y = x.f )</td>
<td>( o_i \in pt(x), \ o_j \in pt(o_i.f) ), ( o_j \in pt(y) )</td>
<td>( y \leftarrow o_i.f )</td>
</tr>
</tbody>
</table>
Pointer Flow Graph: An Example

Program

\[(o_i \in pt(c), o_i \in pt(d))\]

\[
\begin{align*}
a &= b; & ① \\
c.f &= a; & ② \\
d &= c; & ③ \\
c.f &= d; & ④ \\
e &= d.f; & ⑤ \\
\end{align*}
\]

Pointer flow graph

- Variable node \(v\)
- Instance field node \(O_i.f\)

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\[(o_i \in pt(c), o_i \in pt(d))\]

\[\begin{align*}
&\text{a} = \text{b}; \quad ① \\
&\text{c.f} = \text{a}; \quad ② \\
&\text{d} = \text{c}; \quad ③ \\
&\text{c.f} = \text{d}; \quad ④ \\
&\text{e} = \text{d.f}; \quad ⑤
\end{align*}\]

Pointer flow graph

- Variable node \(v\)
- Instance field node \(O_i.f\)

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\((o_i \in pt(c), o_i \in pt(d))\)

1. \(a = b;\)
2. \(c.f = a;\)
3. \(d = c;\)
4. \(c.f = d;\)
5. \(e = d.f;\)

Pointer flow graph

- Variable node
- Instance field node

Tian Tan @ Nanjing University
Program

\[(o_i \in pt(c), o_i \in pt(d))\]

\[\text{a} = \text{b}; \quad \text{①}\]
\[\text{c.f} = \text{a}; \quad \text{②} \]
\[\text{d} = \text{c}; \quad \text{③}\]
\[\text{c.f} = \text{d}; \quad \text{④}\]
\[\text{e} = \text{d.f}; \quad \text{⑤}\]

Pointer flow graph

- Variable node
- Instance field node

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\[(o_i \in pt(c), o_i \in pt(d))\]

\[a = b; \quad \text{(1)}\]
\[c.f = a; \quad \text{(2)}\]
\[d = c; \quad \text{(3)}\]
\[c.f = d; \quad \text{(4)}\]
\[e = d.f; \quad \text{(5)}\]

Pointer flow graph

- Variable node
- Instance field node

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\( (o_i \in pt(c), o_i \in pt(d)) \)

\[
\begin{align*}
\text{a} &= \text{b}; & (1) \\
\text{c.f} &= \text{a}; & (2) \\
\text{d} &= \text{c}; & (3) \\
\text{c.f} &= \text{d}; & (4) \\
\text{e} &= \text{d.f}; & (5)
\end{align*}
\]

Pointer flow graph

- Variable node \( v \)
- Instance field node \( O_i.f \)

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\[(o_i \in pt(c), o_i \in pt(d))\]

\[\begin{align*}
a &= b; & (1) \\
c.f &= a; & (2) \\
d &= c; & (3) \\
c.f &= d; & (4) \\
e &= d.f; & (5)
\end{align*}\]

Pointer flow graph

- Variable node
- Instance field node

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\( (o_i \in pt(c), o_i \in pt(d)) \)

1. \(a = b;\)
2. \(c.f = a;\)
3. \(d = c;\)
4. \(c.f = d;\)
5. \(e = d.f;\)

Pointer flow graph

- Variable node
- Instance field node

Tian Tan @ Nanjing University
Pointer Flow Graph: An Example

Program

\( (o_i \in pt(c), o_i \in pt(d)) \)

\[
\begin{align*}
a &= b; & ① \\
c.f &= a; & ② \\
d &= c; & ③ \\
c.f &= d; & ④ \\
e &= d.f; & ⑤
\end{align*}
\]

Pointer flow graph

- Variable node
- Instance field node

```
Tian Tan @ Nanjing University
```
Pointer Flow Graph: An Example

Program

\[(o_i \in pt(c), o_i \in pt(d))\]

```plaintext
a = b; ①
c.f = a; ②
d = c; ③
c.f = d; ④
e = d.f; ⑤
```

Pointer flow graph

- **Variable node**
- **Instance field node**

Tian Tan @ Nanjing University
With PFG, pointer analysis can be solved by computing *transitive closure* of the PFG
**Pointer Flow Graph: An Example**

Program

\[
(o_i \in pt(c), o_i \in pt(d))
\]

a = b; \hspace{1cm} ①  
c.f = a; \hspace{1cm} ②  
d = c; \hspace{1cm} ③  
c.f = d; \hspace{1cm} ④  
e = d.f; \hspace{1cm} ⑤

Pointer flow graph

- Variable node  
- Instance field node $O_i.f$

With PFG, pointer analysis can be solved by computing **transitive closure** of the PFG.

E.g., e is reachable from b on the PFG, which means that the objects pointed by b may flow to and also be pointed by e.
Pointer Flow Graph: An Example

Program

\[ (o_i \in pt(c), o_i \in pt(d)) \]

\[ j: \quad b = \text{new} \ T(); \]
\[ a = b; \quad ① \]
\[ c.f = a; \quad ② \]
\[ d = c; \quad ③ \]
\[ c.f = d; \quad ④ \]
\[ e = d.f; \quad ⑤ \]

With PFG, pointer analysis can be solved by computing *transitive closure* of the PFG

E.g, \( e \) is reachable from \( b \) on the PFG, which means that the objects pointed by \( b \) may flow to and also be pointed by \( e \)

Tian Tan @ Nanjing University
With PFG, pointer analysis can be solved by computing \textit{transitive closure} of the PFG.

E.g., \texttt{e} is reachable from \texttt{b} on the PFG, which means that the objects pointed by \texttt{b} may flow to and also be pointed by \texttt{e}.
Implementing Pointer Analysis

1. Build pointer flow graph (PFG)

2. Propagate points-to information on PFG
Implementing Pointer Analysis

1. Build pointer flow graph (PFG)

2. Propagate points-to information on PFG

Mutually dependent
Implementing Pointer Analysis

1. Build pointer flow graph (PFG)

2. Propagate points-to information on PFG

Program

\((o_i \in pt(c), o_i \in pt(d))\)

\(a = b;\)

\(c.f = a;\)

\(d = c;\)

\(c.f = d;\)

\(e = d.f;\)

Mutually dependent

Pointer flow graph

\(a\)

\(b\)

\(c\)

\(d\)

\(e\)

\(O_i.f\)
Implementing Pointer Analysis

1. Build pointer flow graph (PFG)

Mutually dependent

2. Propagate points-to information on PFG

Program

\( o_i \in pt(c), o_i \in pt(d) \)

\[
\begin{align*}
  &a = b; &1 \\
  &c.f = a; &2 \\
  &d = c; &3 \\
  &c.f = d; &4 \\
  &e = d.f; &5
\end{align*}
\]

Pointer flow graph

PFG is dynamically updated during pointer analysis