软件分析

南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾 谭添

Static Program Analysis

Pointer Analysis

Nanjing University

Tian Tan

2021

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

```
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

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

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

```
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

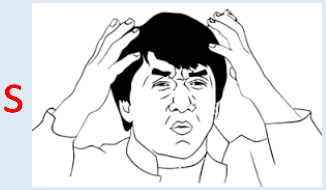
Problem of CHA

```
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

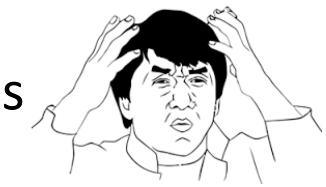
Via Pointer Analysis

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

n points to new One

```
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

Via Pointer Analysis

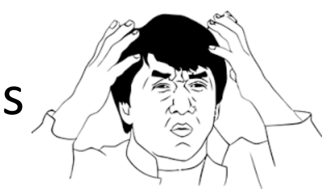
```
void foo() {
    Number n = new One();
    int x = n.get();
}
```

n points to new One

```
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

Constant propagation

- x = 1

Via Pointer Analysis

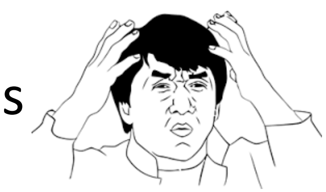
```
void foo() {
    Number n = new One();
    int x = n.get();
}
```

n points to new One

```
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

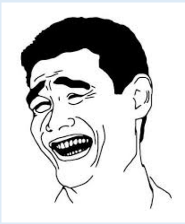


Constant propagation

- x = NAC imprecise

Pointer analysis: based on **points-to relation**

- 1 call target
- 0 false positive



Constant propagation

- x = 1 precise

Contents

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

- William E. Weihl, "*Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables*". POPL 1980.

Still an active area today

- OOPSLA'18, FSE'18, TOPLAS'19, OOPSLA'19, TOPLAS'20, OOPSLA'21 …

Example

"Which objects a pointer can point to?"

Program

Points-to relations

```
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

```
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

Variable	Object
a	new A
x	new B

Example

"Which objects a pointer can point to?"

Program

```
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

Variable	Object
a	new A
x	new B
this	?
b	?

Example

"Which objects a pointer can point to?"

Program

```
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

Variable	Object
a	new A
x	new B
this	new A
b	new B

Example

"Which objects a pointer can point to?"

Program

```
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

Variable	Object
a	new A
x	new B
this	new A
b	new B

Field	Object
new A.b	new B

Example

"Which objects a pointer can point to?"

Program

```
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

Variable	Object
a	new A
x	new B
this	new A
b	new B
y	?

Field	Object
new A.b	new B

Example

"Which objects a pointer can point to?"

Program

```
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; }
}
```

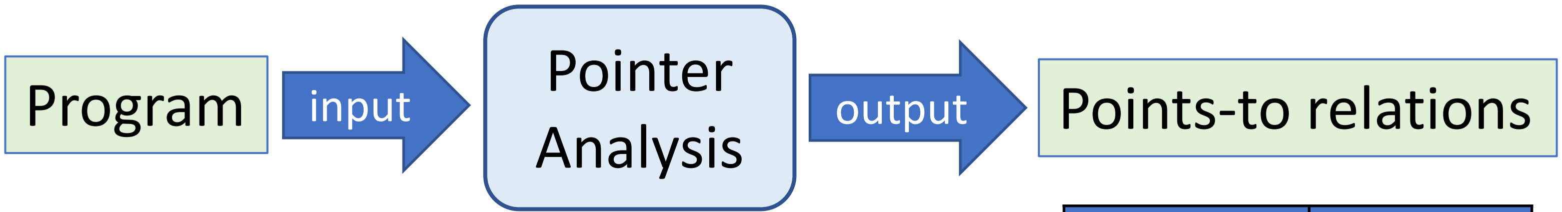
Points-to relations

Variable	Object
a	new A
x	new B
this	new A
b	new B
y	new B

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; }
}
```

Variable	Object
a	new A
x	new B
this	new A
b	new B
y	new B

Field	Object
new A.b	new B

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 = 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 = new C();
q = p;
x = new X();
y = 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
  - Call graph, aliases, …
- Compiler optimization
  - Virtual call inlining, …
- Bug detection
  - Null pointer detection, …
- Security analysis
  - Information flow analysis, …
- And many more …

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

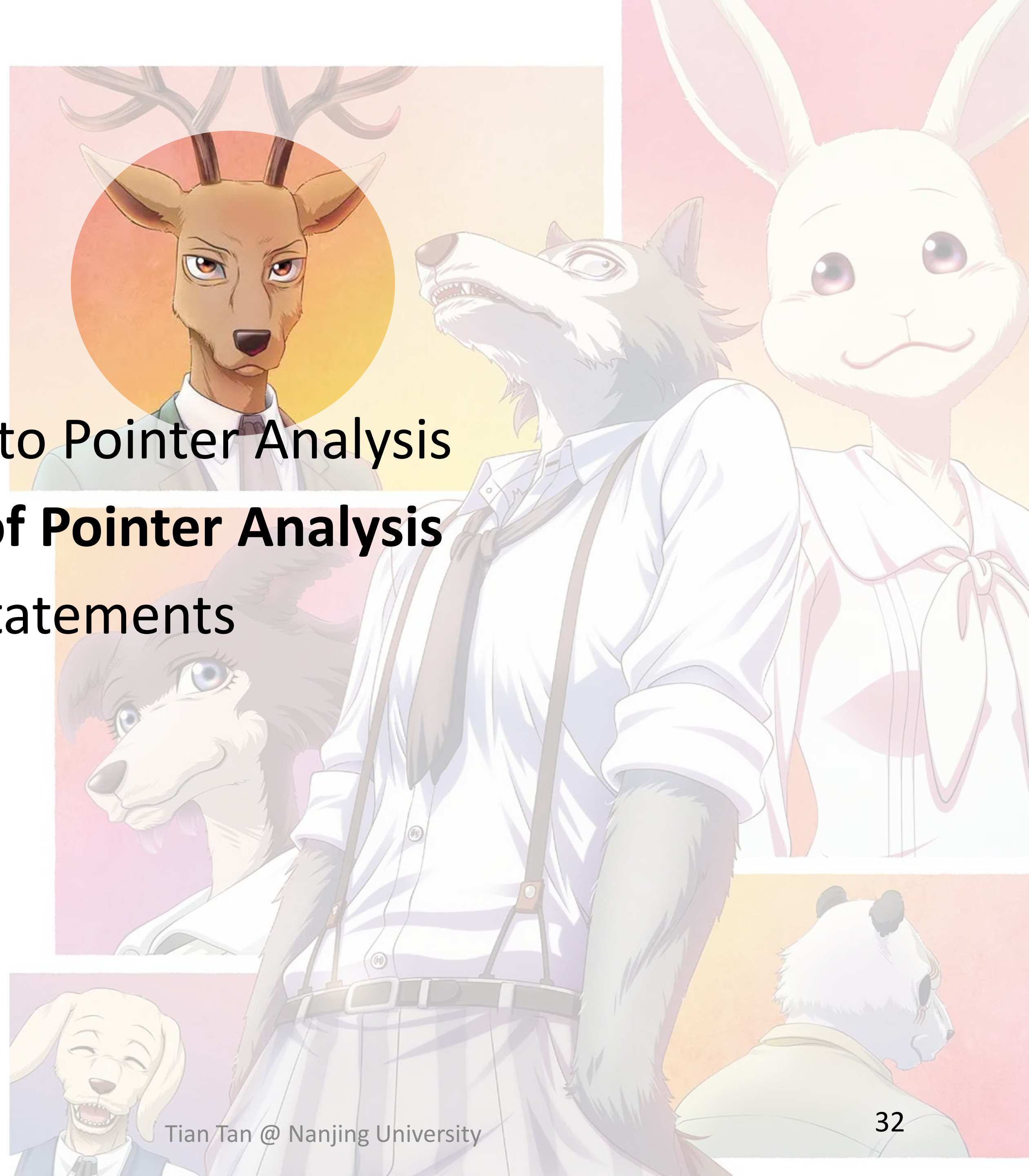
Applications of Pointer Analysis

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

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

*Pointer Analysis - Report from Dagstuhl Seminar 13162. 2013.

Contents



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

Factor	Problem	Choice
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

Factor	Problem	Choice
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

Factor	Problem	Choice
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

Factor	Problem	Choice
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

Factor	Problem	Choice
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

Heap Abstraction

How to model heap memory?

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

```
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

```
for (…) {
  A a = new A();
}
```

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

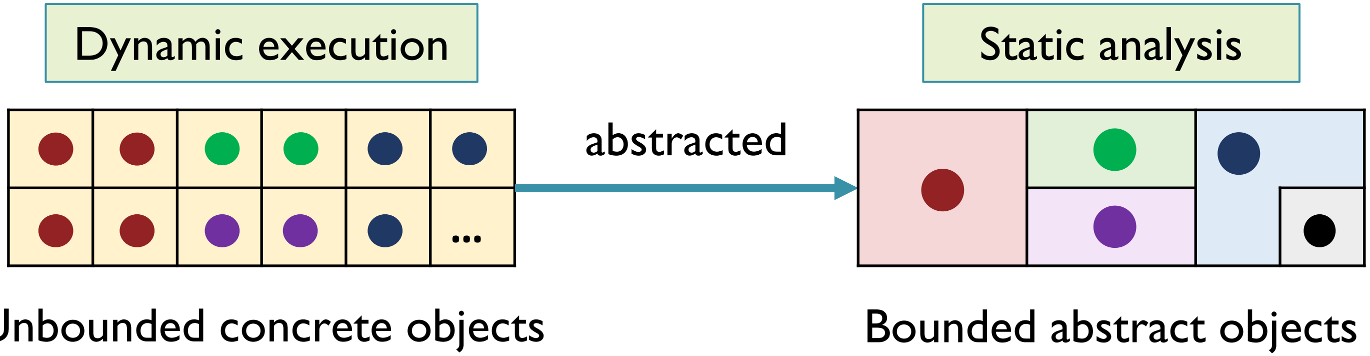
Heap Abstraction

How to model heap memory?

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

```
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

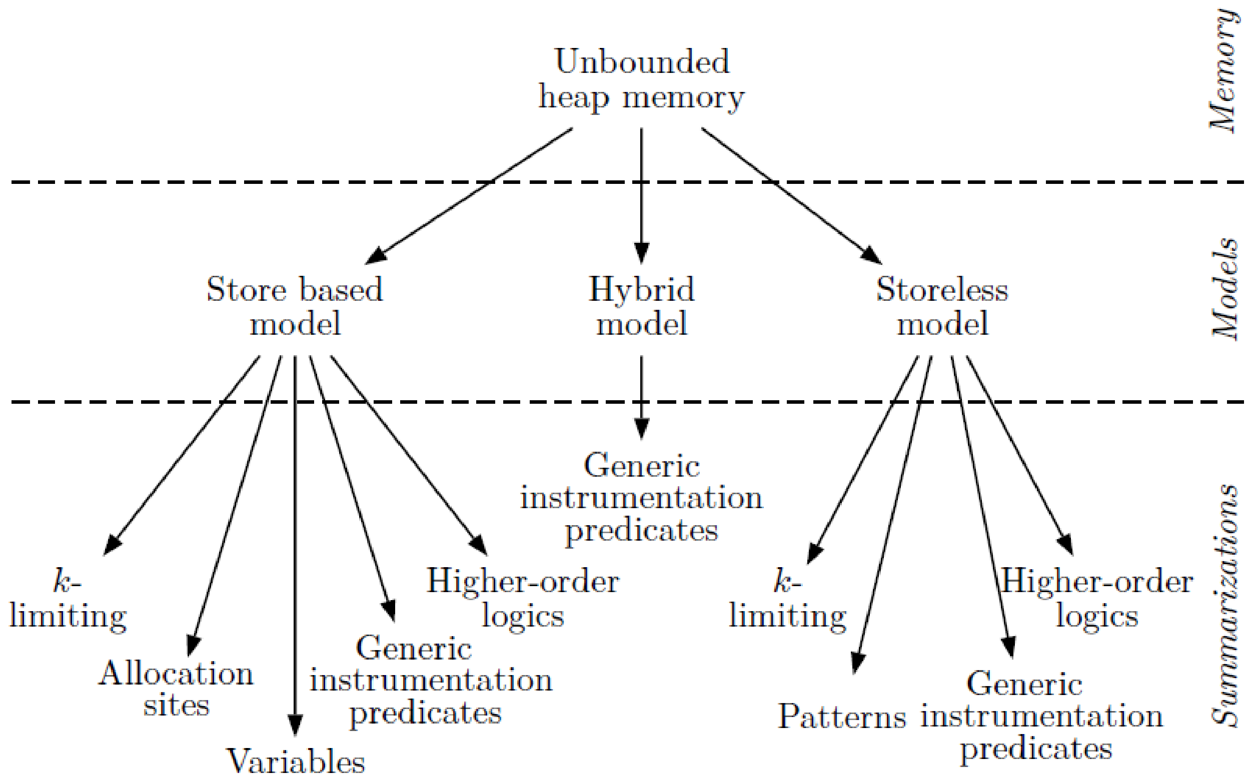


Figure 2. Heap memory can be modeled as storeless, store based, or hybrid. These models are summarized using allocation sites, k-limiting, patterns, variables, other generic instrumentation predicates, or higher-order logics.

Vini Kanvar, Uday P. Khedker, "*Heap Abstractions for Static Analysis*". ACM CSUR 2016

Heap Abstraction

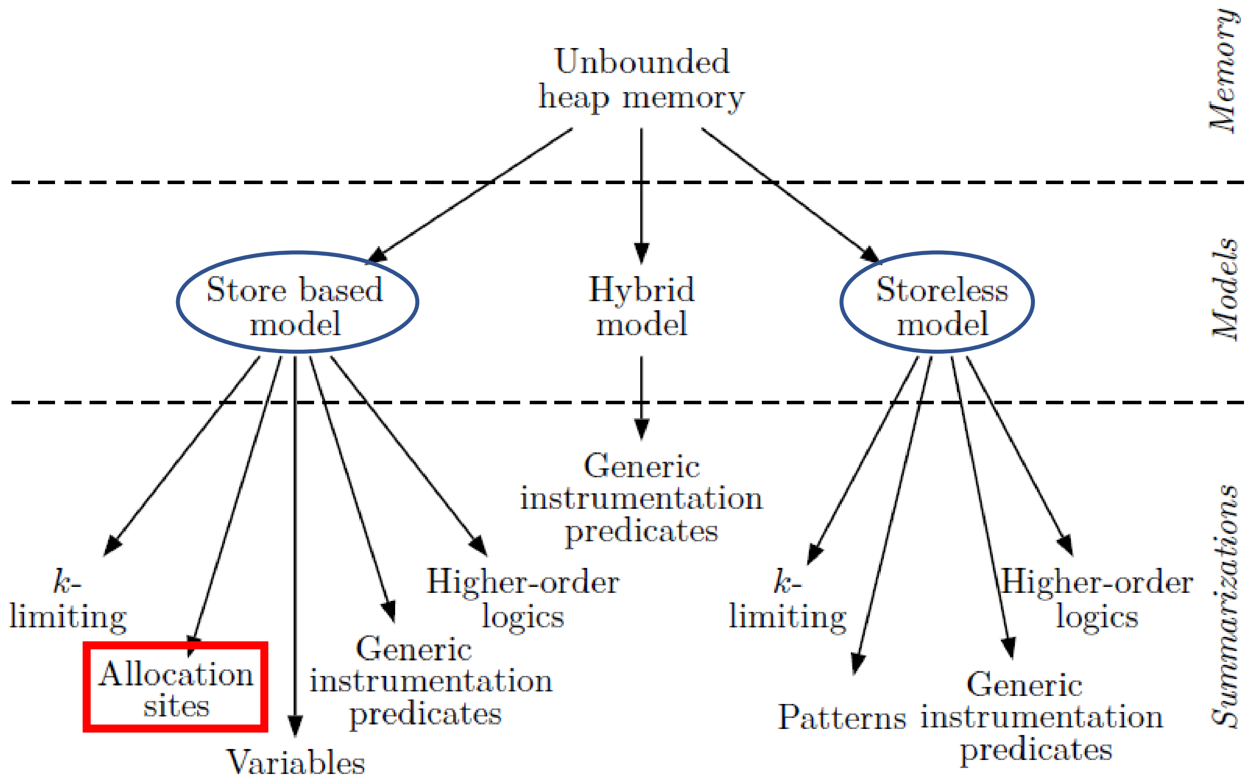


Figure 2. Heap memory can be modeled as storeless, store based, or hybrid. These models are summarized using allocation sites, k-limiting, patterns, variables, other generic instrumentation predicates, or higher-order logics.

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

```
for (i = 0; i < 3; ++i) {
    a = new A();
    …
}
```

o_2, iteration `i = 0`
o_2, iteration `i = 1`
o_2, iteration `i = 2`

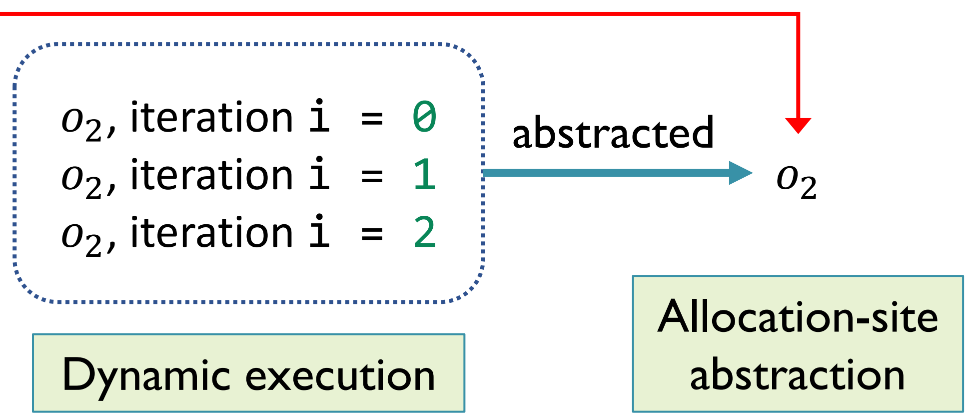
Dynamic execution

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

```
for (i = 0; i < 3; ++i) {
    a = new A();
    …
}
```



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

```
for (i = 0; i < 3; ++i) {
    a = new A();
    …
}
```

o_2, iteration i = 0
o_2, iteration i = 1
o_2, iteration i = 2

Dynamic execution

abstracted → o_2

Allocation-site abstraction

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

Key Factors in Pointer Analysis

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

Factor	Problem	Choice
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

Context Sensitivity

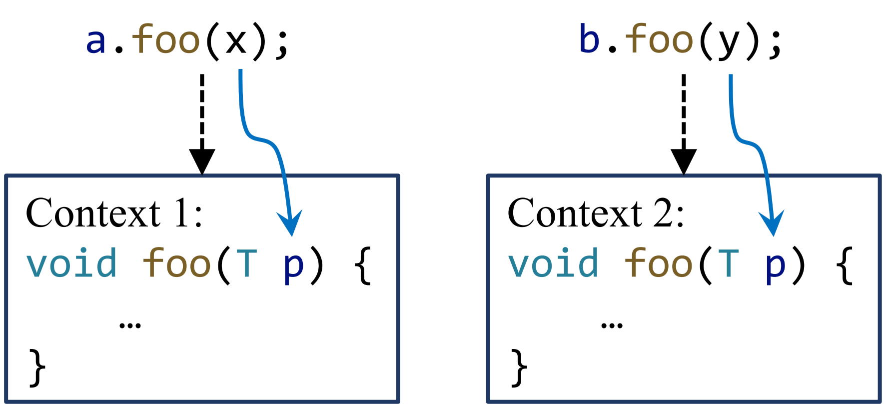
How to model calling contexts?

Context-sensitive	Context-insensitive
Distinguish different calling contexts of a method	Merge all calling contexts of a method
Analyze each method multiple times, once for each context	Analyze each method once

Context Sensitivity

How to model calling contexts?

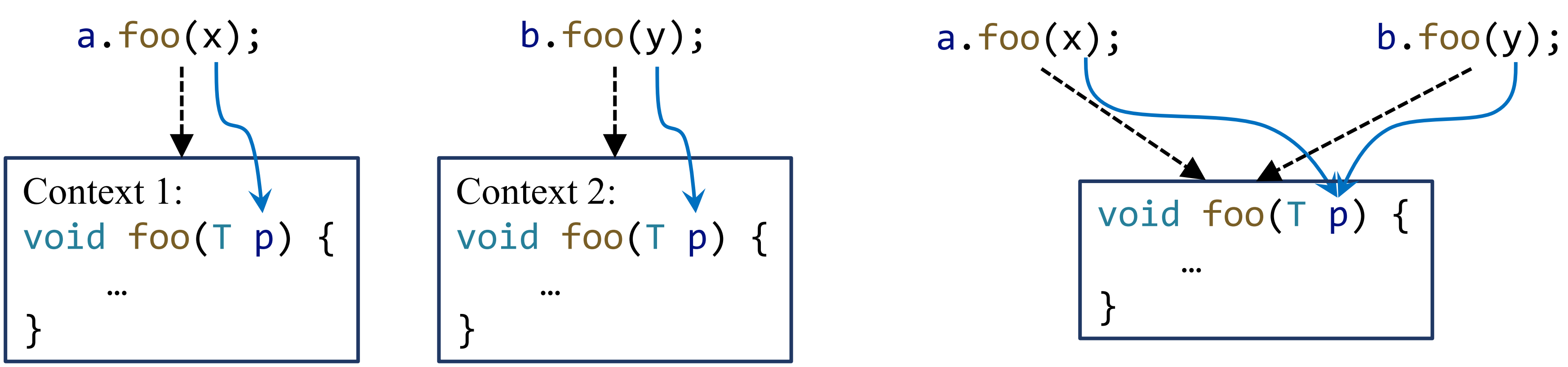
Context-sensitive	Context-insensitive
Distinguish different calling contexts of a method	Merge all calling contexts of a method
Analyze each method multiple times, once for each context	Analyze each method once



Context Sensitivity

How to model calling contexts?

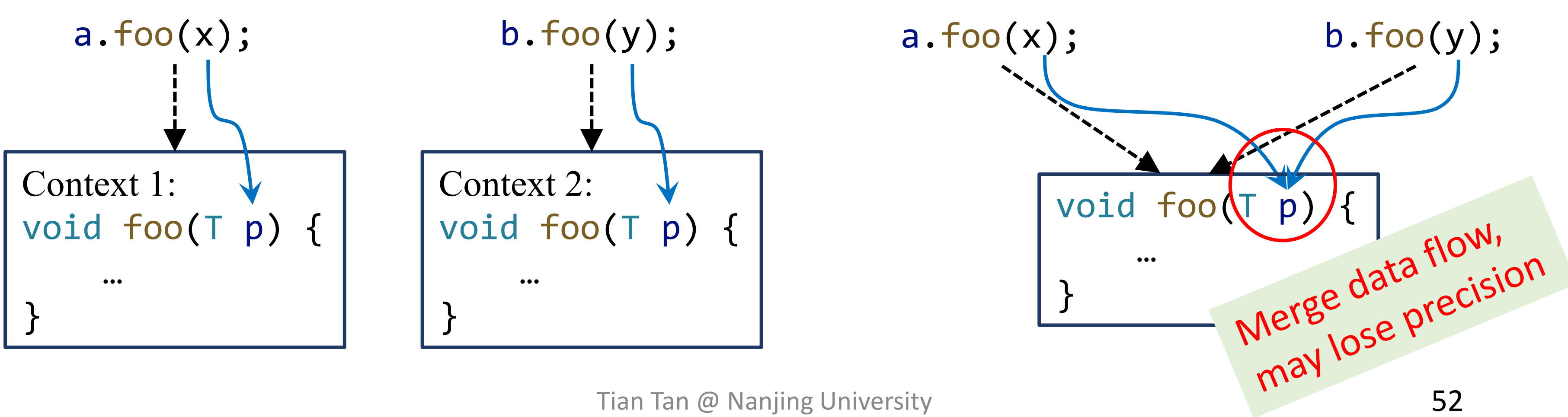
Context-sensitive	Context-insensitive
Distinguish different calling contexts of a method	Merge all calling contexts of a method
Analyze each method multiple times, once for each context	Analyze each method once



Context Sensitivity

How to model calling contexts?

Context-sensitive	Context-insensitive
Distinguish different calling contexts of a method	Merge all calling contexts of a method
Analyze each method multiple times, once for each context	Analyze each method once



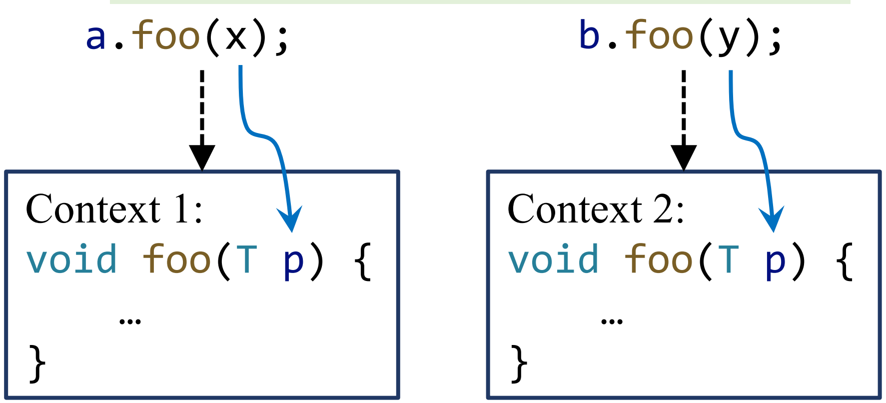
Context Sensitivity

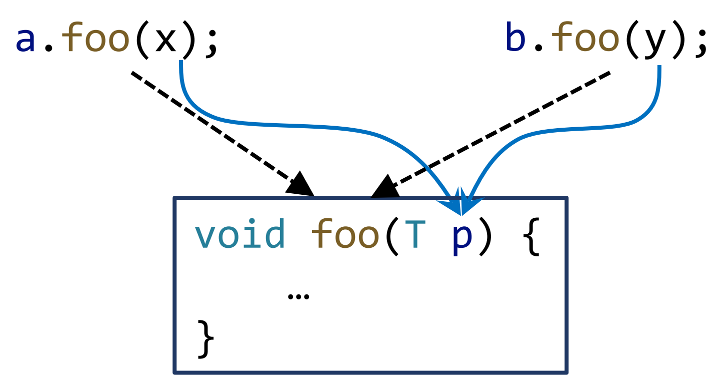
How to model calling contexts?

Context-sensitive	Context-insensitive
Distinguish different calling contexts of a method	Merge all calling contexts of a method
Analyze each method multiple times, or	Analyze each method once

Very useful technique
Significantly improve precision
More details in **later lectures**

We start with **this**





Key Factors in Pointer Analysis

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

Factor	Problem	Choice
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

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

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

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

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

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

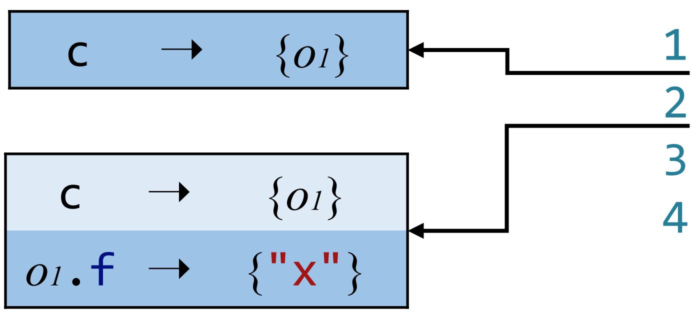
c $\rightarrow$ $\{o_1\}$

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

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

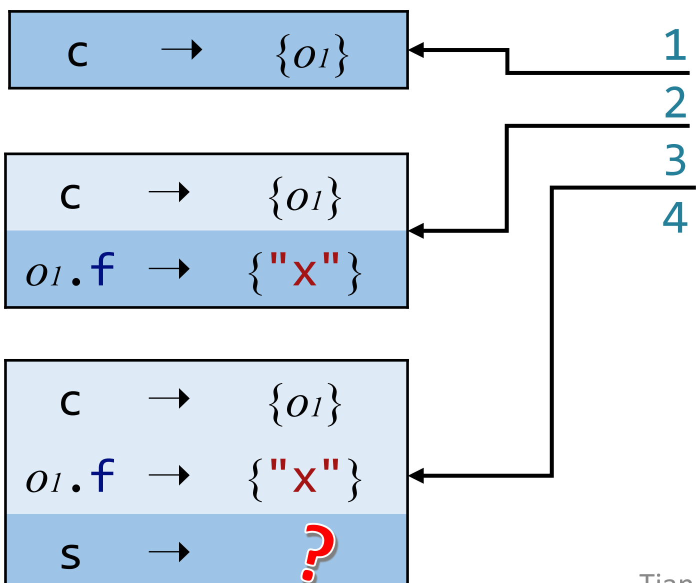



```
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

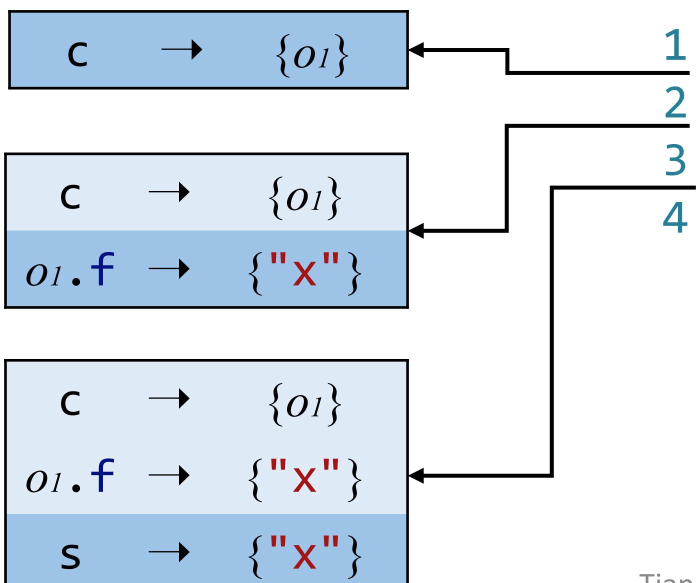



```
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

Flow Sensitivity

How to model control flow?

Flow-sensitive	**Flow-insensitive**
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program

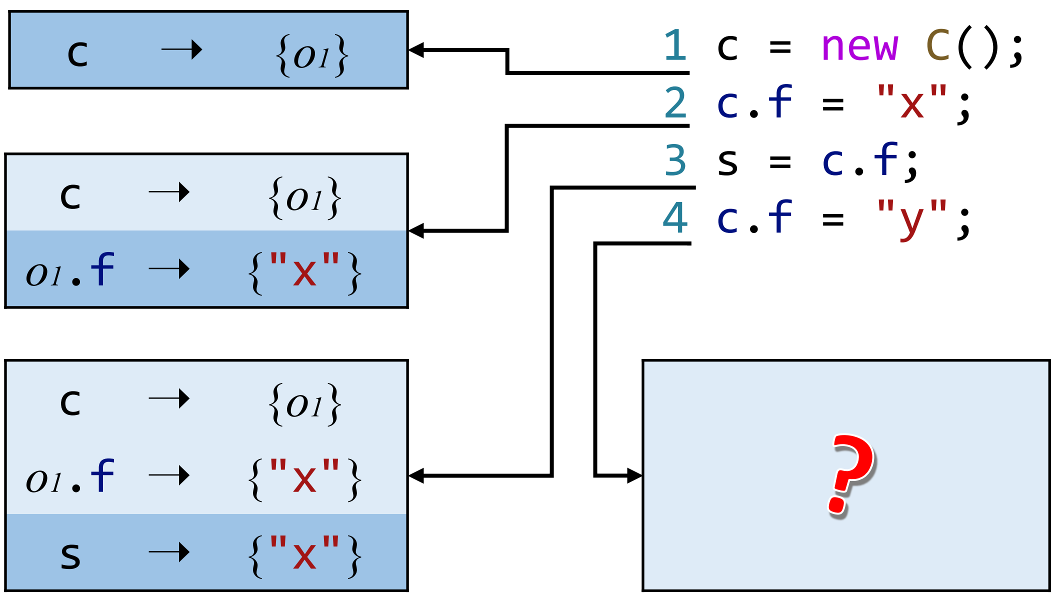



```
c = new C();
c.f = "x";
s = c.f;
c.f = "y";
```

Flow Sensitivity

How to model control flow?

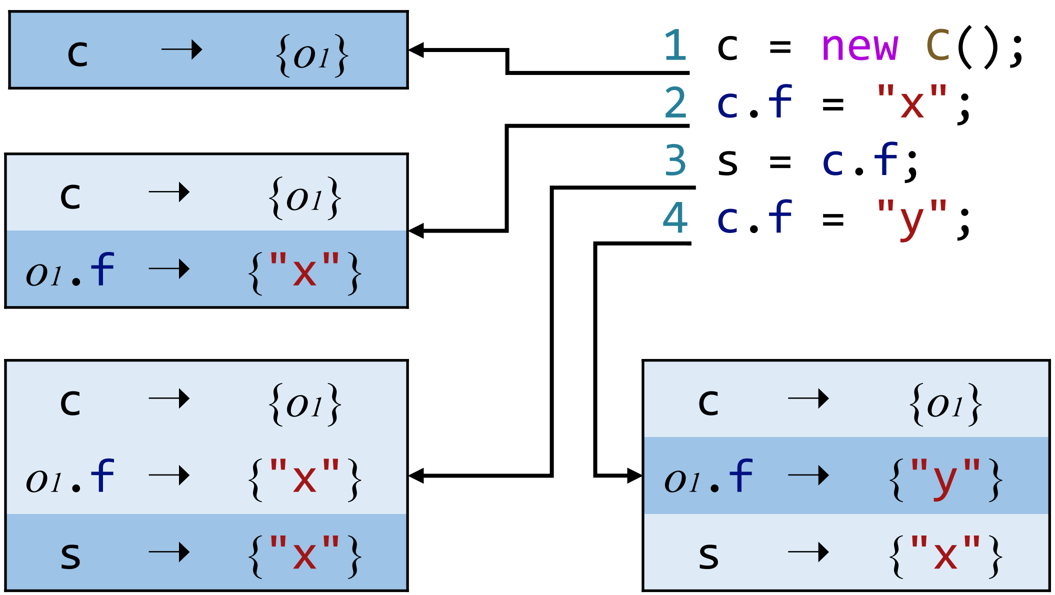
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

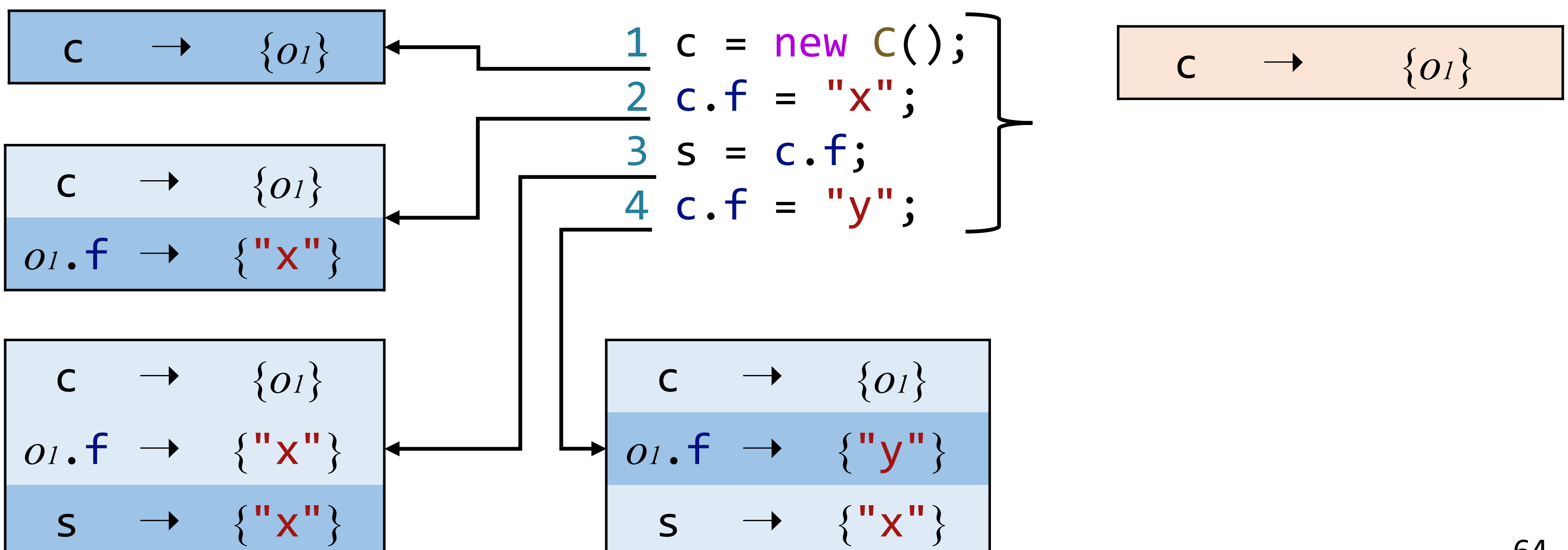
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

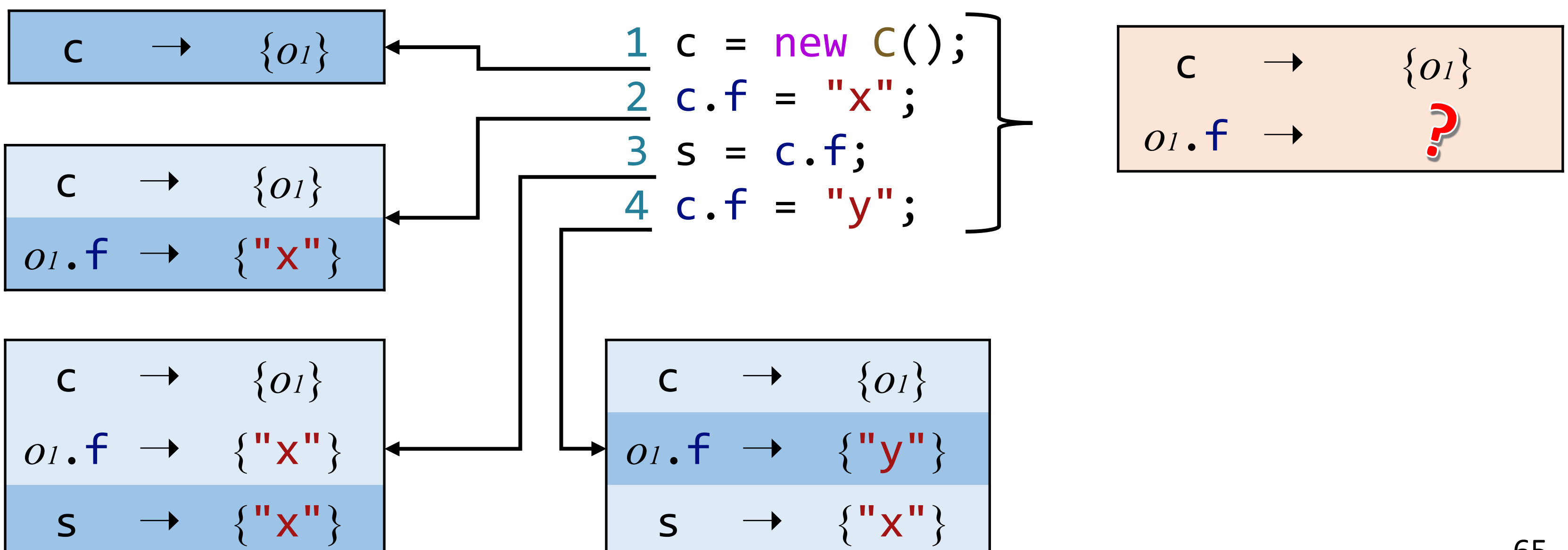
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

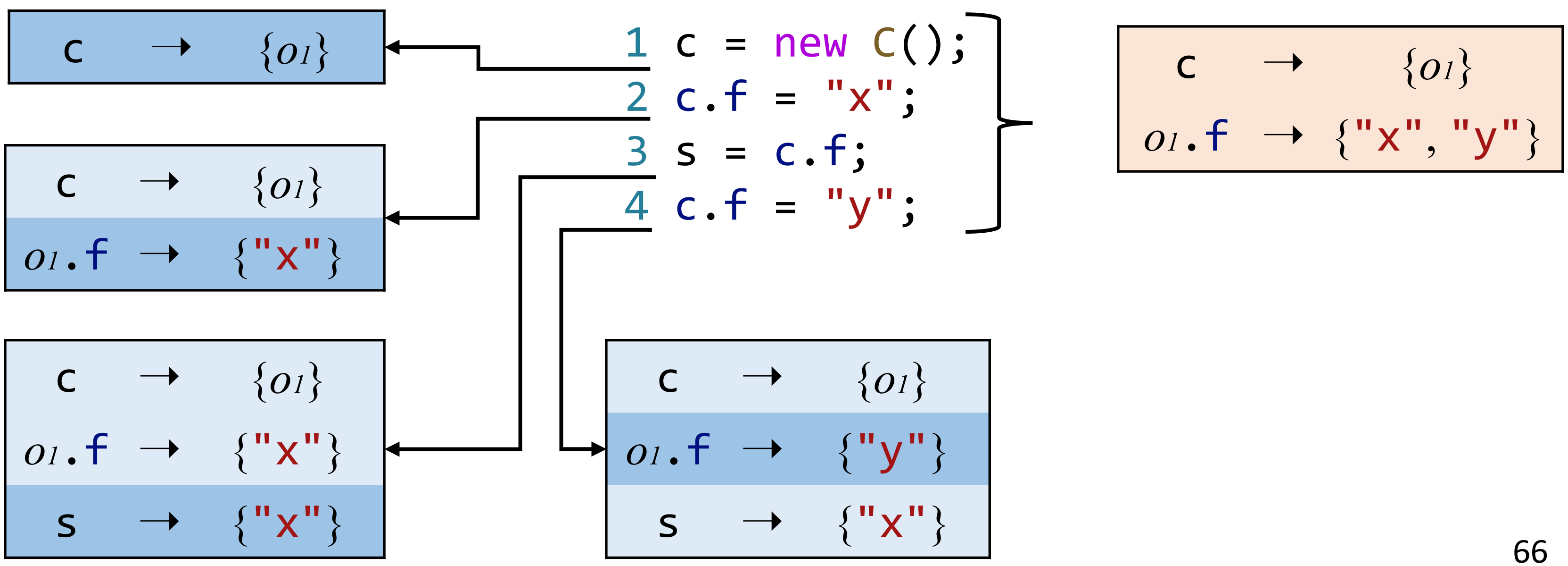
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

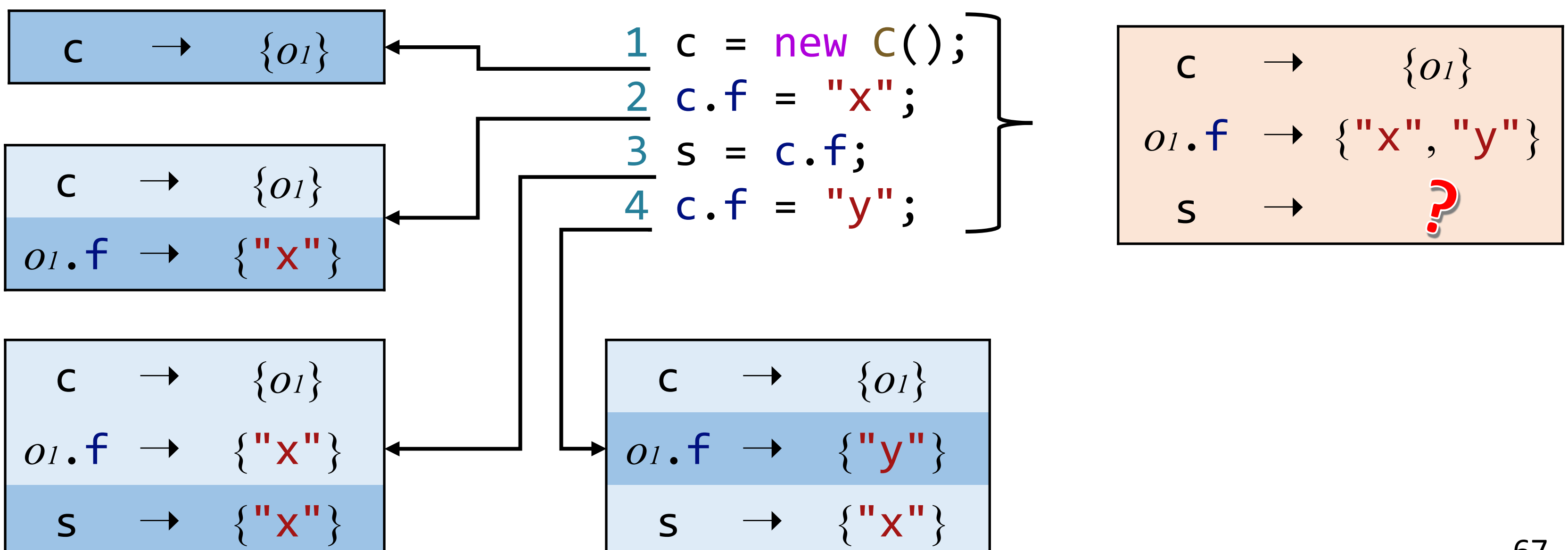
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

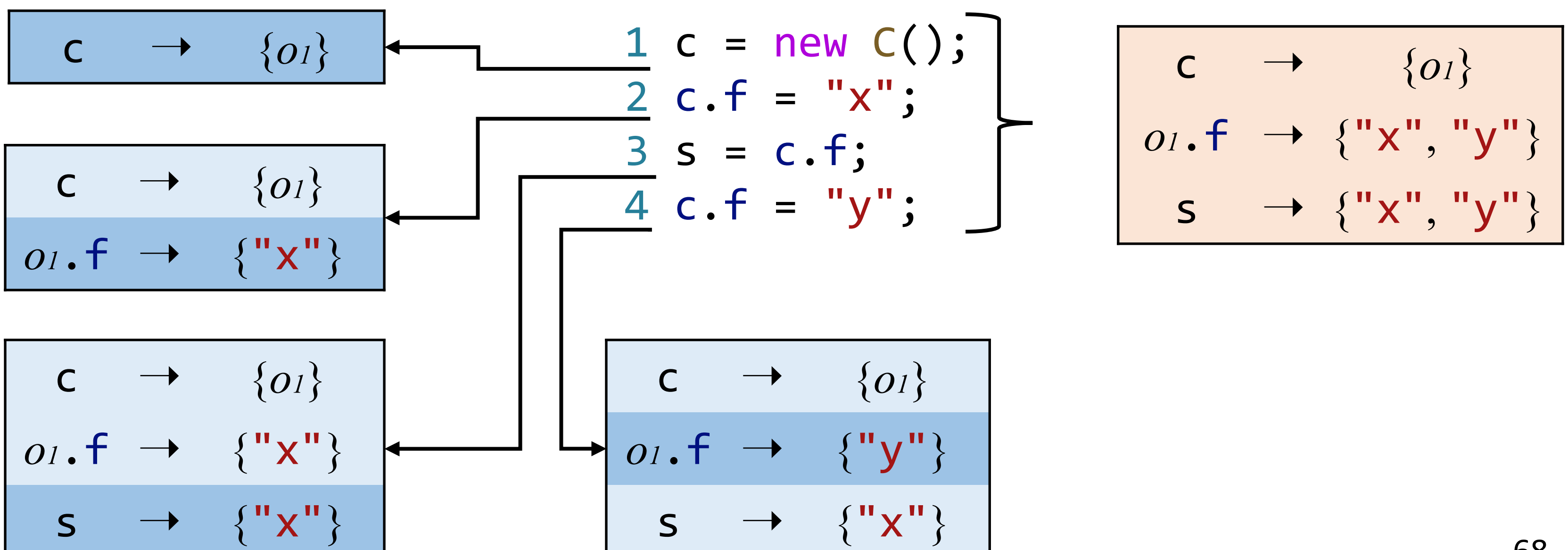
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

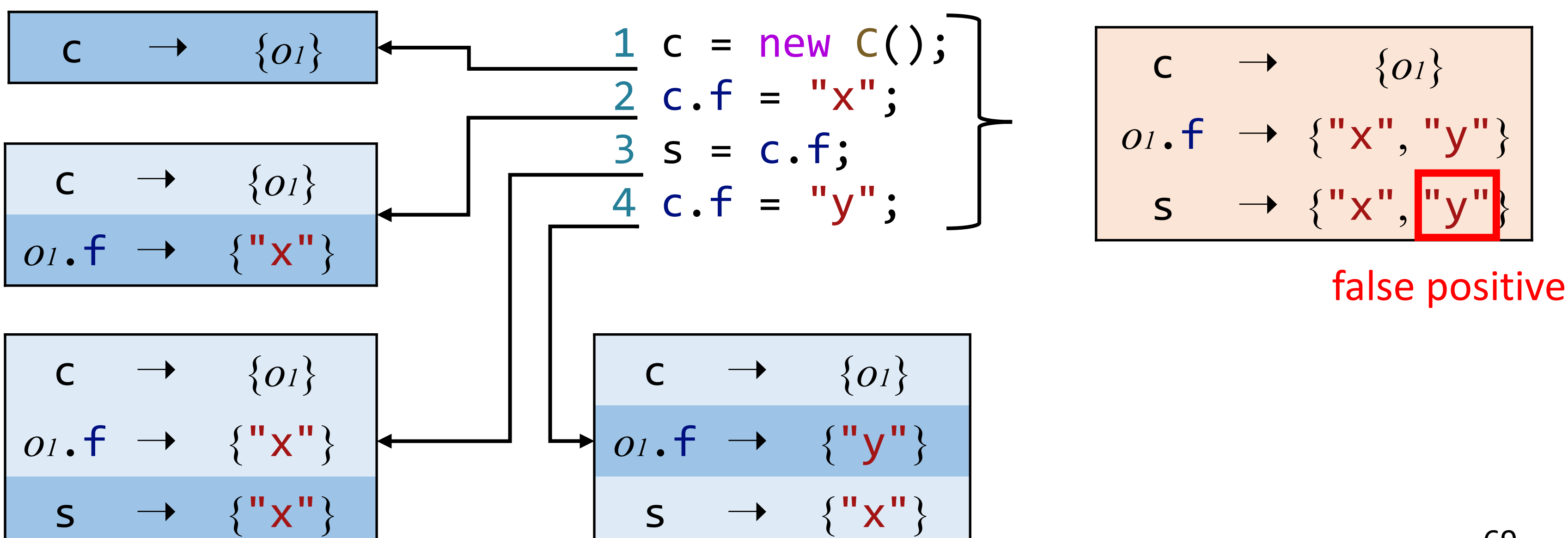
Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

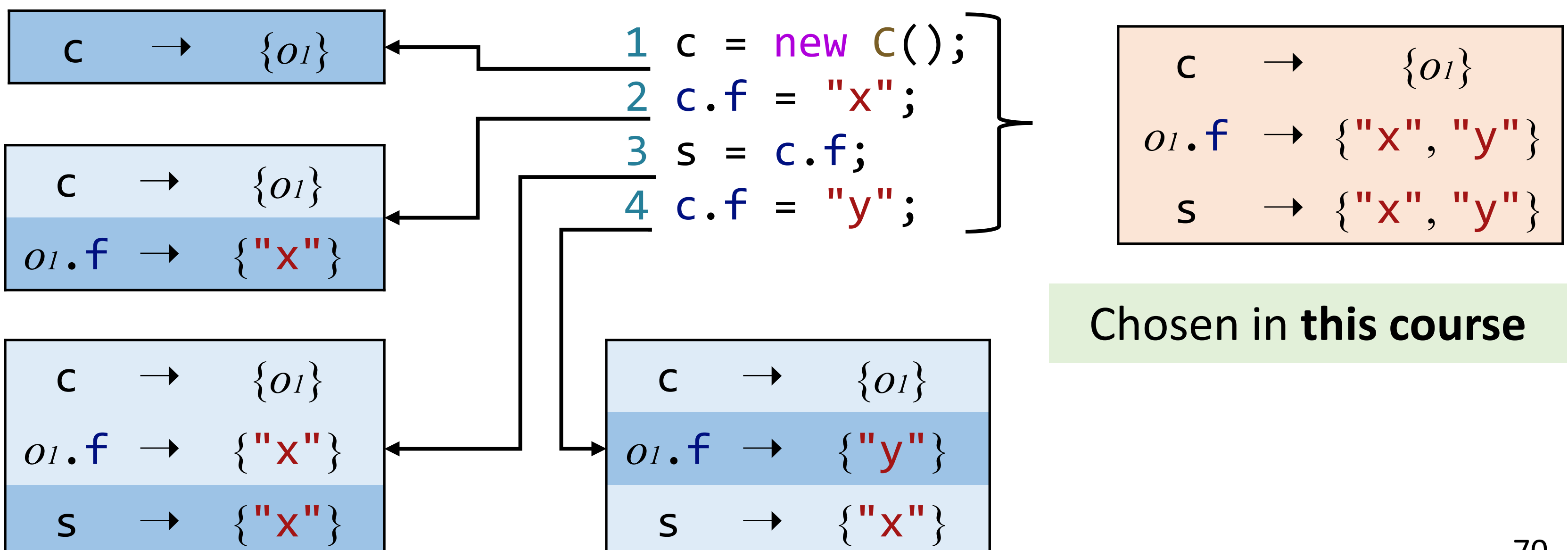
Flow-sensitive	**Flow-insensitive**
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Flow Sensitivity

How to model control flow?

Flow-sensitive	Flow-insensitive
Respect the execution order of the statements	Ignore the control-flow order, treat the program as a set of unordered statements
Maintain a map of points-to relations at each program location	Maintain one map of points-to relations for the whole program



Key Factors in Pointer Analysis

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

Factor	Problem	Choice
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

Analysis Scope

Which parts of program should be analyzed?

Whole-program	Demand-driven
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

Analysis Scope

Which parts of program should be analyzed?

Whole-program	**Demand-driven**
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

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

Analysis Scope

Which parts of program should be analyzed?

Whole-program	Demand-driven
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

x	$\rightarrow$	$\{o_1\}$
y	$\rightarrow$	$\{o_1\}$
z	$\rightarrow$	$\{o_4\}$

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

Analysis Scope

Which parts of program should be analyzed?

Whole-program	Demand-driven
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

x	→	$\{o_1\}$
y	→	$\{o_1\}$
z	→	$\{o_4\}$

```
1 x = new A();
2 y = x;
3 …
4 z = new T();
5 z.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?

Whole-program	Demand-driven
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

x	$\rightarrow$	$\{o_1\}$
y	$\rightarrow$	$\{o_1\}$
z	$\rightarrow$	$\{o_4\}$

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

z	$\rightarrow$	$\{o_4\}$

Client: call graph construction
Site of interest: line 5

Analysis Scope

Which parts of program should be analyzed?

Whole-program	**Demand-driven**
Compute points-to information for all pointers in the program	Only compute points-to information for the pointers that may affect specific sites of interest (on demand)
Provide information for all possible clients	Provide information for specific clients

Chosen in **this course**

x	→	$\{o_1\}$
y	→	$\{o_1\}$
z	→	$\{o_4\}$

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

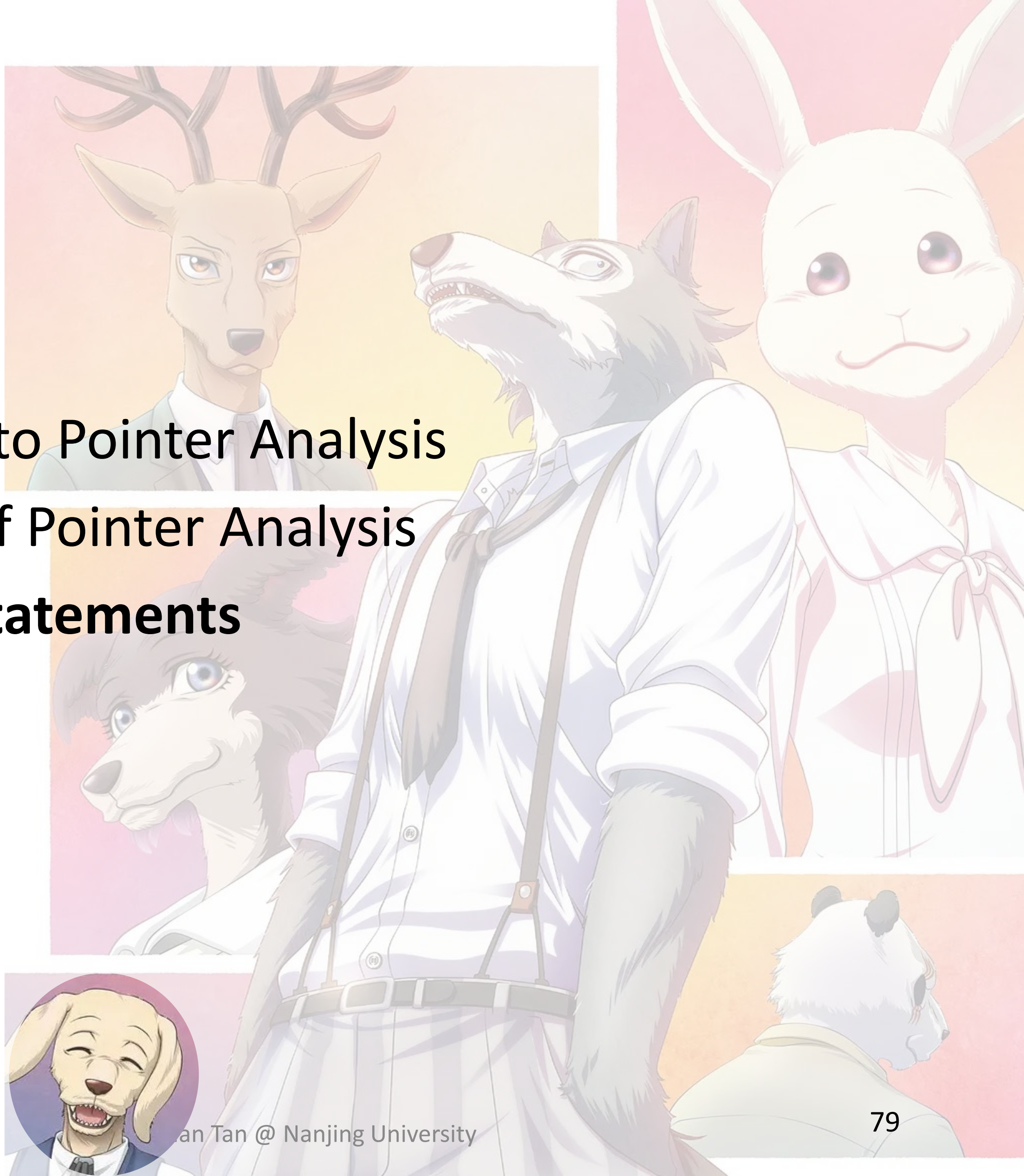
z	→	$\{o_4\}$

Client: call graph construction
Site of interest: line 5

Pointer Analysis in This Course

Factor	Problem	Choice
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

Contents



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~~
  - …

Do not directly affect pointers
Ignored in pointer analysis

- We only focus on **pointer-affecting statements**

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` ⟵ Modeled as an object (pointed by `x`) with a field `f`
- Array element: `array[i]`

Pointers in Java

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

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

Real code

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

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

Pointer-Affecting Statements

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

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 = 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()`

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 X 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

Static Program Analysis

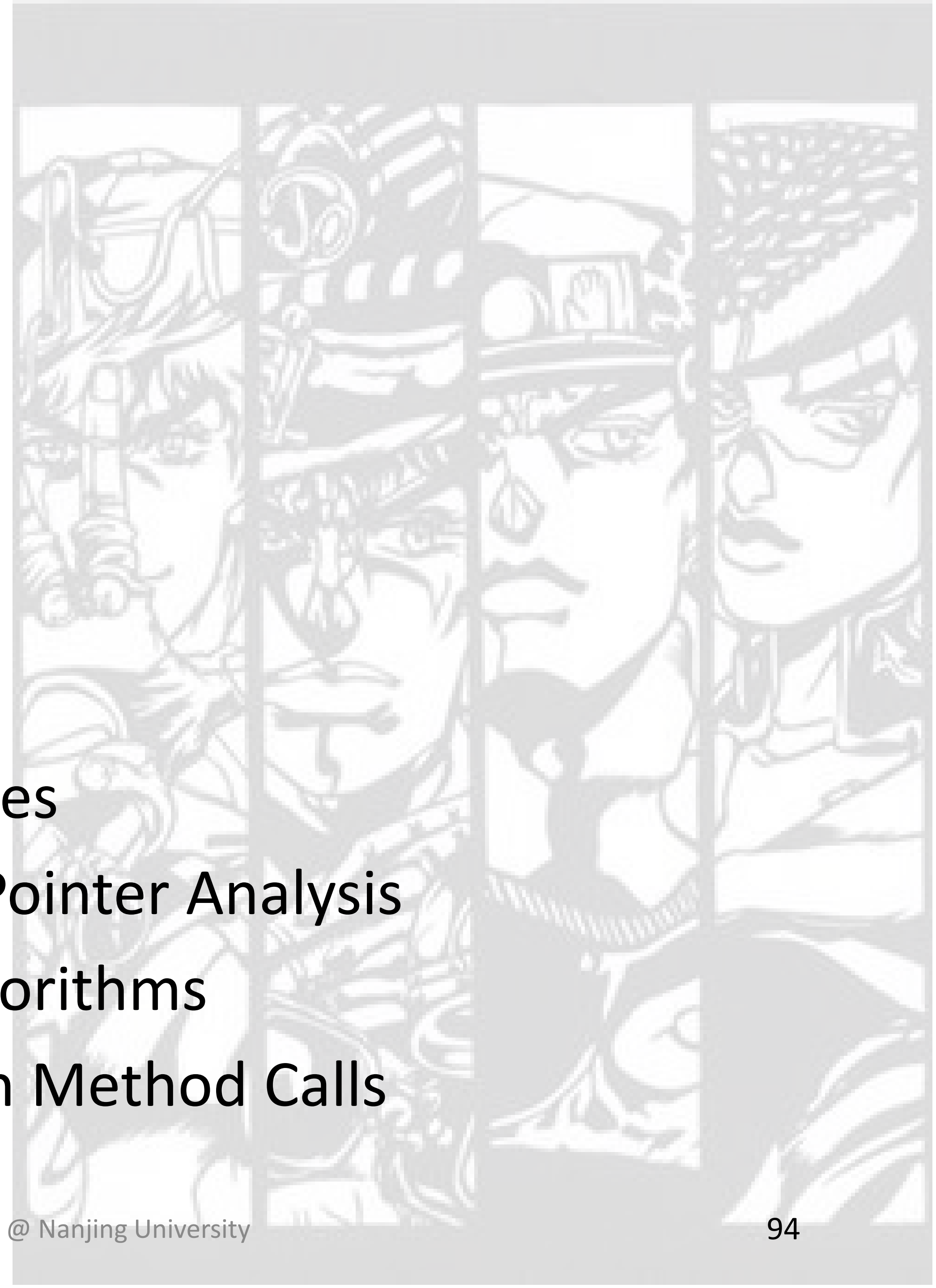
Pointer Analysis
Foundations (I)

Nanjing University

Tian Tan

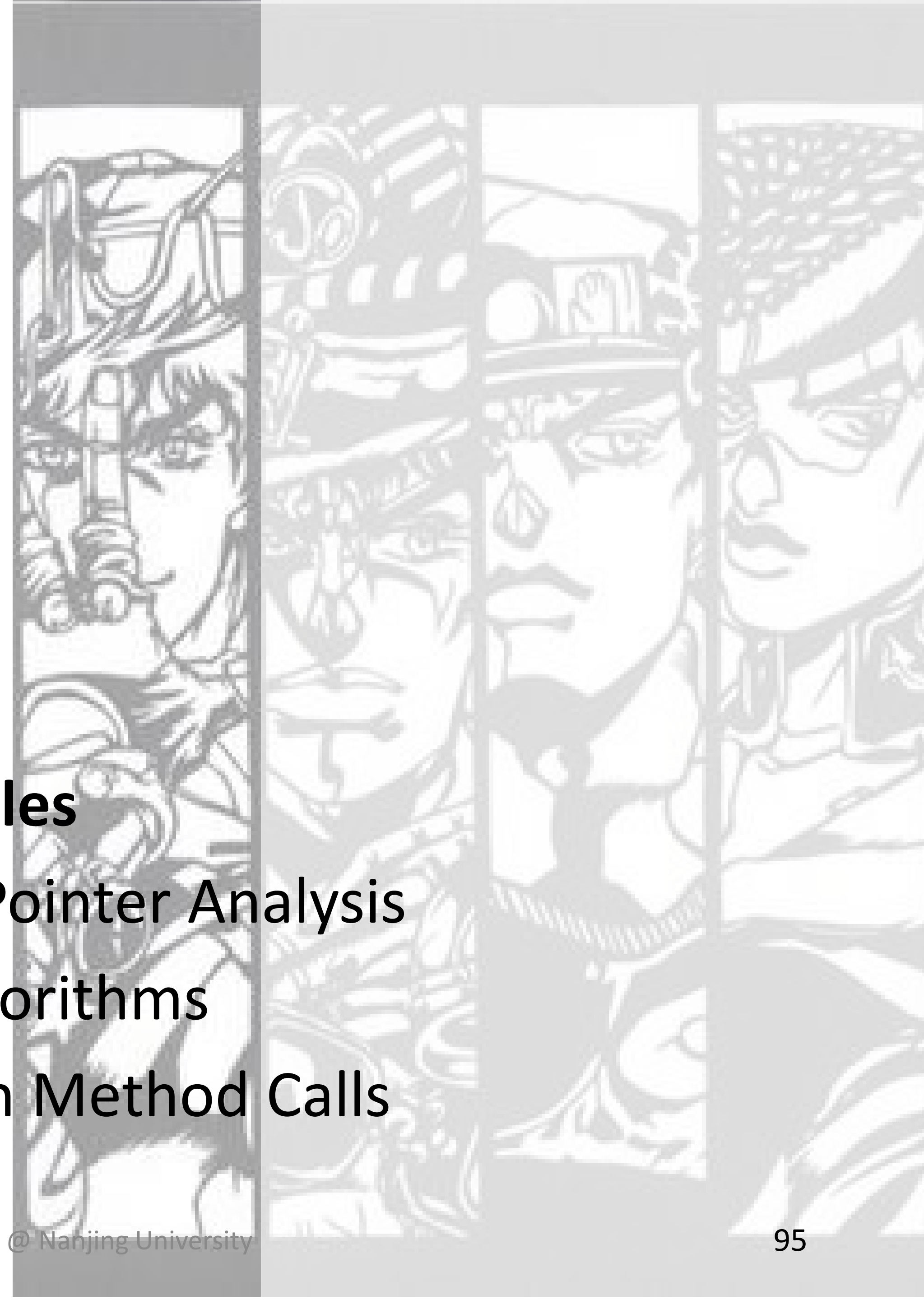
2020

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 = new T()`	First focus on these statements (suppose the program has just one method)
Assign	`x = y`	
Store	`x.f = y`	
Load	`y = x.f`	
Call	`r = x.k(a, …)`	← Will come back to this in pointer analysis with method calls

Domain and Notations

Variables:	$x, y \in \mathrm{V}$
Fields:	$f, g \in \mathrm{F}$
Objects:	$o_i, o_j \in \mathrm{O}$
Instance fields:	$o_i.f, o_j.g \in \mathrm{O} \times \mathrm{F}$
Pointers:	$\mathrm{Pointer} = \mathrm{V} \cup (\mathrm{O} \times \mathrm{F})$
Points-to relations:	$pt : \mathrm{Pointer} \rightarrow \mathcal{P}(\mathrm{O})$

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

Rules

Kind	Statement	Rule
New	`i: x = new T()`	$\dfrac{}{o_i \in pt(x)}$
Assign	`x = y`	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\dfrac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\dfrac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$

Rules

Kind	Statement	Rule
New	`i: x = new T()`	$\frac{}{o_i \in pt(x)}$ ← unconditional
Assign	`x = y`	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$ ← premises ← conclusion
Store	`x.f = y`	$\frac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\frac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$

Rule: New

$$\frac{}{o_i \in pt(x)}$$

⟶ Conclusion

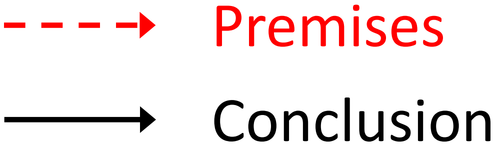
o_i

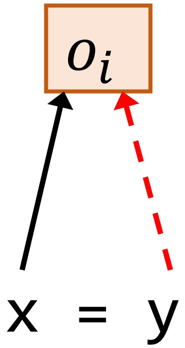
↑

```
i: x = new T()
```

Rule: Assign

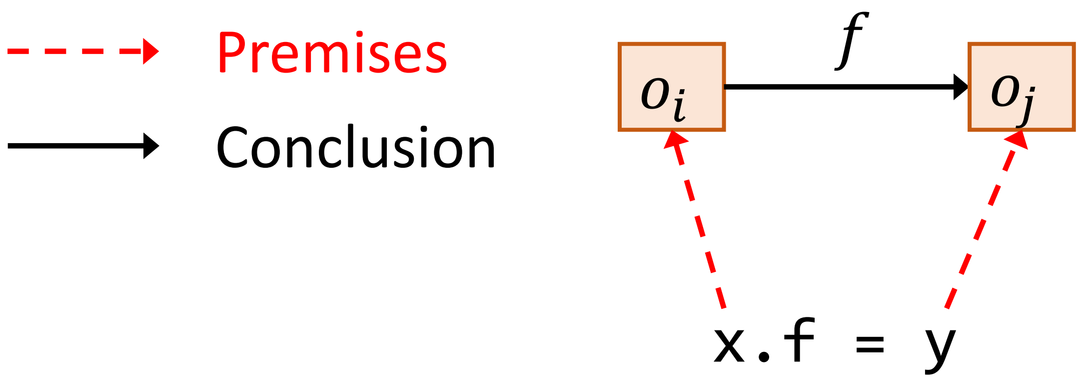
$$\frac{o_i \in pt(y)}{o_i \in pt(x)}$$





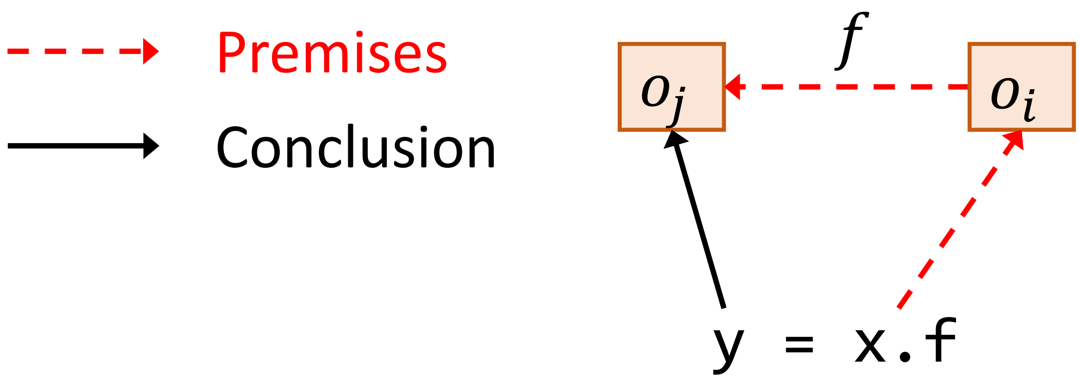
Rule: Store

$$\frac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$$



Rule: Load

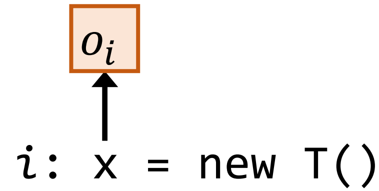
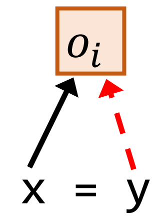
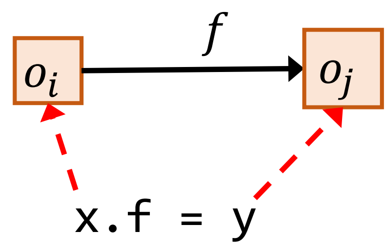
$$\frac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$$



Rules

Premises

Conclusion

Kind	Rule	Illustration
New	$\dfrac{}{o_i \in pt(x)}$	
Assign	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$	
Store	$\dfrac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$	
Load	$\dfrac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$