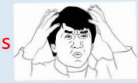
一、引入

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

仅基于 declared Type 信息
指定初始点
↓
不精确

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; }
}
```

基于 RO 的实例信息
精确判定

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

目标：Pointer 指向的 Memory Location

⟹ 对 OOPL，计算 Pointer（variable 或 field）指向的 object.

Ps: PTA是 may-Analysis，也即指针可能指向的 object 集都算。

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

关键点在于利用控制流/数据流信息，精确求出 Pointer 所指向的 object 的类型，而不再是简单使用 declared Type 进行近似估计，这样对于 Call Graph 的求解会准确很多。

☆常见的 PTA 应用

Pointer Analysis Vs. Alias Analysis

↓ 判断各 Pointer 指向哪个 object？

↳ 判断两个 Pointer 是否指向同一个 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

Alias Analysis是PTA的子问题

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.

二、PTA的组成

PTA是一个非常复杂的系统，有很多能够影响Precision和efficiency的关键因素

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

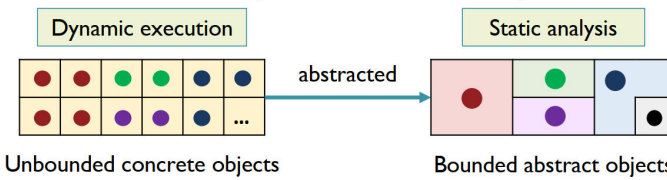
(1) Heap abstraction

在程序运行时，其可能会向堆区动态申请内存，而在逻辑上，堆区是没有边界约束的，因此在循环和递归的场景下，Heap object的数量可能是无穷的，直接PTA则会无法终止，因此我们需要合适的对堆的抽象。

- 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



主流堆抽象方法：

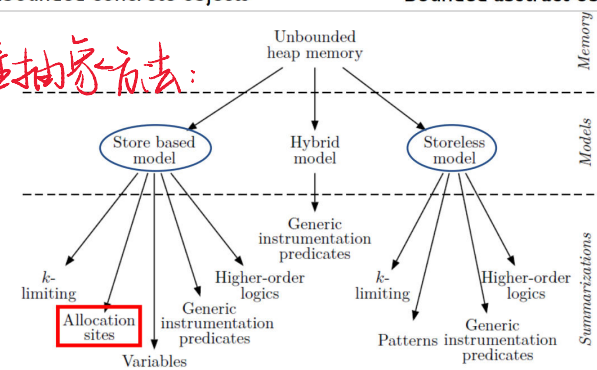


Figure 2. Heap memory can be modeled as storeless, store based, or hybrid. These models are summarized using allocation sites, *k*-limiting, patterns, variables, other generic instrumentation predicates, or higher-order logics.

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

我们只介绍 Allocation Sites，当下最常见的方法：

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

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

Dynamic execution

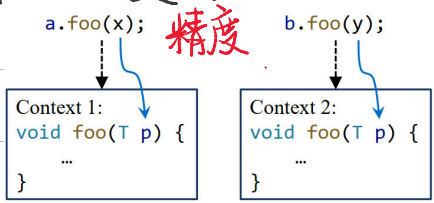
Allocation-site abstraction

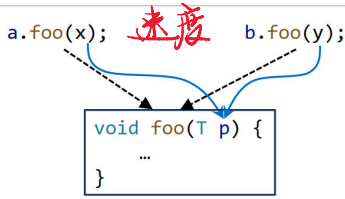
The number of allocation sites in a program is bounded, thus the abstract objects must be finite.

就是循环迭代多少次，一个 Allocation Site 对应一个 Heap Object.

2、Context Sensitivity

相同方法在不同的程序点调用，其对应的 Context 是不同的，而 Context 又是方法运行的信息之源，因此选择合适的上下文建模方式是十分关键的。





Context-Sensitive：为同一方法的不同调用各自抽象出不同的上下文，针对各个调用的数据流

进行单独处理。

Context-Insensitive: Merge 所有不同调用的 dataflow 于一个 context 中，同一方法的所有不同调用都基于这个上下文。因此每个方法实际上只需要分析一次。

(3) Flow Sensitivity

实际上就是是否利用程序有序性所带来的信息

Flow-Sensitive

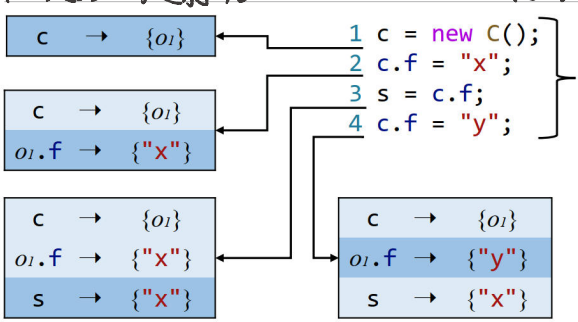
程序=语句的有序链

每个程序点都维护一张指向关系映射表

Flow-Insensitive

程序=语句的无序集

为整个程序维护一张指向关系映射表



Ps: 对于 C/C++，我们发现 Flow-Sensitive 带来的精度提升显著提高了分析结果，而在 Java 中没有证据表明 Flow-Sensitive 的效果明显更好，因此我们一般使用后者。

(4) Analysis Scope

Whole Program: 计算程序中的所有指向关系

Demand Driven: 根据需求计算程序中的部分指向关系。

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

Ps: 由于后者的运行实际上也需要许多其它语句信息的支持，所以不见得一定比整个程序一起分析慢多少。加之不同任务的分析范围可能会有重叠，所以不如整个程序一起分析。

三、Concerned Statements

现在我们已经大致明白PTA的指导原则了，包括怎样进行抽象，同一方法不同调用的上下文如何进行处理，是否利用流信息，分析的范围是什么。但是我们却一直没有深入研究指针分析的对象究竟是什么？反正不能是程序中的所有语句吧！

- 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**

什么是"Pointer?"　　怎么样叫"affecting"?

1. 详解Pointer

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

(1) Local variable: x

普通的局部变量，指向一片内存区域。

(2) Static field: C.f

类成员，像一种"全局变量"，分析原理与上面的局部变量相同。

(3) Instance field: x.f

实例成员变量，与局部变量最大的不同在于其加载于堆空间，而 LV 加载于栈空间（栈序加载时就被分配完毕了，这其实说明了为什么加载于静态域的类成员变量与其拥有相同的分析逻辑）

(4) Array Element: array[i]

数组内存空间同样动态分配在堆中，由于加上索引计算其具体指向的位置是相当困难的，因此我们一般忽略索引信息，将数组抽象为分配在堆中的单一域，类似 Instance field.

```
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

2. 影响 Pointer 的语句

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()`

由于多态的存在，前两者的指向关系分析是简单的，主要依旧在 virtual call。

Ps：涉及实例成员的语句可能会有复杂的内存处理过程，但好在我们的分析是基于三地址码的，中间过程都以临时变量的形式存储起来了，因此我们关注主点即可。

```
x.f.g.h = y;
```

↓

```
t1 = x.f
t2 = t1.g
t2.h = y;
```