Context-Sensitivity 是运用于跨方法分析的一个非常基础的技术。

一、Motivation

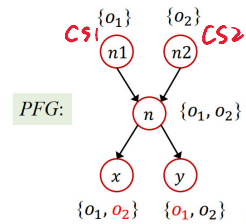
Problem of Context-Insensitive Pointer Analysis

```
void main() {
  Number n1, n2, x, y;
  n1 = new One(); // o1
  n2 = new Two(); // o2
  x = id(n1);
  y = id(n2);
  int i = x.get();
}
Number id(Number n) {
  return n;
}

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

Context insensitivity, `x.get()`:
- 2 call targets
- 1 false positive

Constant propagation
- `i` = NAC



要得到i的结果 ⟶ 求解x.get()的target method

↓

源于id()的返回值 ⟵ 对x进行指针分析

但是在CIS的PTA中，我们处理方法调用时是直接构建cs处的参数/返回值与target method的参数/返回值间的PFG边，也即各cs处指向信息会在目标方法处汇聚，各个调用的上下文信息没有进行区分。

Context-Sensitivity ⇒ 对同一目标方法的不同调用进行隔离

Via Context-Sensitive Pointer Analysis

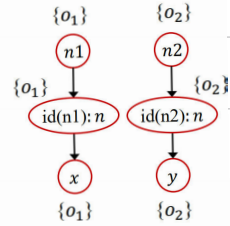
```
void main() {
  Number n1, n2, x, y;
  n1 = new One(); // o1
  n2 = new Two(); // o2
  x = id(n1);
  y = id(n2);
  int i = x.get();
}
Number id(Number n) {
  return n;
}

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

Context sensitivity, `x.get()`:
- 1 call targets
- 0 false positive

Constant propagation
- i = 1

PFG:



13

二、调用上下文隔离

1) C.I & C.S. 对比

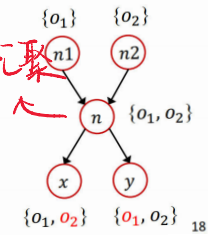
C.I. 由于不同调用(虽然目标方法相同)的指向信息混在一起，这些信息也会通过返回值和过程中的副作用进行扩散，导致大量误报

Imprecision of Context Insensitivity (C.I.)

- In dynamic execution, a method may be called multiple times under different calling contexts
- Under different calling contexts, the variables of the method may point to different objects
- In C.I. pointer analysis, objects under different contexts are mixed and propagated to other parts of program (through return values or side-effects), causing spurious data flows

```
x = id(n1);
y = id(n2);
int i = x.get();

Number id(Number n) {
  return n;
}
```



18

Context-Sensitivity 虽然可能会影响效率，但是会极大地改变精度。

核心之处在于针对同一方法的不同调用的数据流是否分开处理

(2) 调用上下文的建模

由上，那么应该如何标识不同调用的数据流呢？简单处理，直接使用 call-site 进行区分。

- Context sensitivity models calling contexts by distinguishing different data flows of different contexts to improve precision

- The oldest and best-known context sensitivity strategy is call-site sensitivity (call-string)
  - Which represents each context of a method as a chain of call sites, i.e.,
    a call site of the method,
    a call site of the caller,
    a call site of caller of caller, etc.

You will see other variants of context sensitivity in the next lecture

```
1 x = id(n1);
2 y = id(n2);
3 int i = x.get();
4
5 Number id(Number n) {
6   return n;
7 }
```

In call-site sensitivity, method id(Number) has two contexts: [1] and [2]

标识变量具体属于哪个数据流/上下文。

(3) 机制

复制一份完整的 method 5 variables 承载数据流的具体载体，给每次调用所产生的不同的上下文/数据流。数据流 = 上下文(调用值) + 变量

Cloning-Based Context Sensitivity

The most straightforward approach to implement context sensitivity

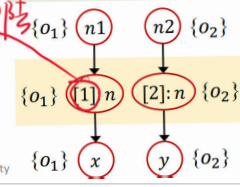
- In cloning-based context-sensitive pointer analysis, each method is qualified by one or more contexts
- The variables are also qualified by contexts (inherited from the method they are declared in)
- Essentially each method and its variables are cloned, one clone per context

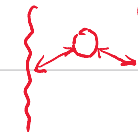
```
1 x = id(n1);
2 y = id(n2);
3 int i = x.get();
4
5 Number id(Number n) {
6   return n;
7 }
```

In call-site sensitivity, method id(Number) has two contexts: [1] and [2]

采用 Call-String 进行标识

知道从哪来、往哪去





三、堆对象上下文敏感

上面隔离上下文不同的调用似乎已经可以做到不同数据流之间的独立性了，但是由于我们抽象堆对象的方法（allocation-site），数据流依旧会在一定程度上产生汇聚。

同一创建点在不同调用之下所产生的对象在语义上应当是不同的

```
X newX(Y y) {
  X x = new X();
  x.f = y;
  return x;
}
```

↓

不同对象会被不同的数据流所操纵

↓

将堆对象依据allocation-site抽象为一个会导致数据流在涉及堆对象的地方产生部分的归并

为了把不同数据流所对应的堆对象区分开，我们应当对不同调用的数据流进行标记，依旧采用 Call-String 建模即可。

Context-Sensitive Heap

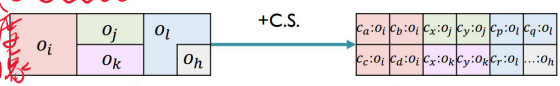
频繁、操纵堆

- OO programs (e.g., Java) are typically heap-intensive
- In practice, to improve precision, context sensitivity should also be applied to heap abstraction
  - The abstract objects are also qualified by contexts (called heap contexts)

The most common choice is to inherit contexts from the method where the object is allocated

  - Context-sensitive heap abstraction provides a finer-grained heap model over allocation-site abstraction

更细粒度的堆抽象模型



实际上就是把不同调用上的标记赋进堆抽象中。

Context-Sensitive Heap: Example

```
1  n1 = new One();
2  n2 = new Two();
3  x1 = newX(n1);
4  x2 = newX(n2);
5  n = x1.f;
6
7  X newX(Number p) {
8    X x = new X();
9    x.f = p;
10   return x;
11 }
12 class X {
13   Number f;
14 }
```

C.S., no C.S. heap

Variable	Object
n1	o_1
n2	o_2
3:p	o_1
3:x	o_8
x1	o_8
4:p	o_2
4:x	o_8
x2	o_8
n	o_1, o_2
Field	**Object**
o_8.f	o_1, o_2

Spurious data flow due to lack of C.S. heap

C.S. + C.S heap

Variable	Object
n1	o_1
n2	o_2
3:p	o_1
3:x	$3{:}o_8$
x1	$3{:}o_8$
4:p	o_2
4:x	$4{:}o_8$
x2	$4{:}o_8$
n	o_1
Field	**Object**
$3{:}o_8$.f	o_1
$4{:}o_8$.f	o_2

Context-sensitive heap improves precision

Ps: C.I. + C.S. heap 的效果实际上不好，C.S. 与 C.S. heap 等一起使用才能发挥更好的作用。

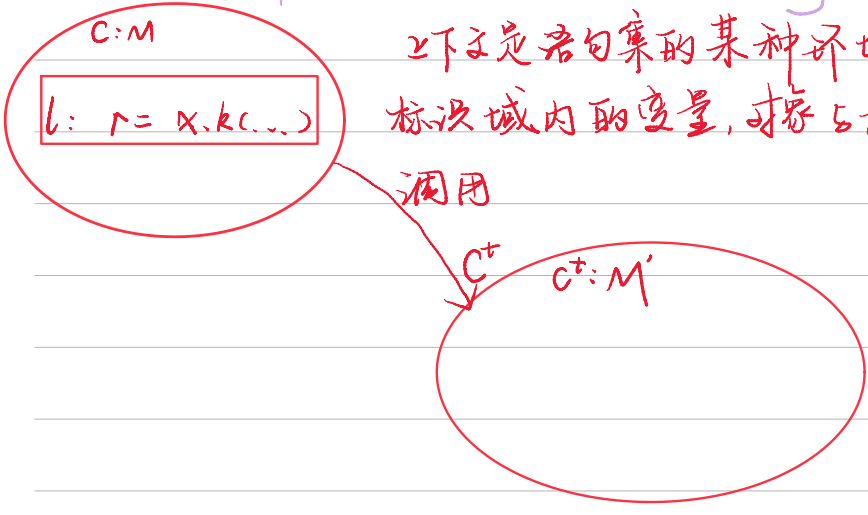
四、C.S. PTA 的规则

1. Domain & Notations

In context-sensitive analysis,
program elements are qualified by **contexts**

Context:	c, c', c''	$\in C$
Context-sensitive methods:	$c:m$	$\in C \times M$
Context-sensitive variables:	$c:x, c':y$	$\in C \times V$
Context-sensitive objects:	$c:o_i, c':o_j$	$\in C \times O$
Fields:	f, g	$\in F$
Instance fields:	$c:o_i.f, c':o_j.g$	$\in C \times O \times F$
Context-sensitive pointers:	CSPointer =	$(C \times V) \cup (C \times O \times F)$
Points-to relations:	$pt:$	$\text{CSPointer} \rightarrow \mathcal{P}(C \times O)$

Ps: C.S 在原有域的基础上增加 Context 标记，主要针对（method + variables）& objects



上下文总是句柄的某种环境，标识域内的变量，对象与方法调用

2. Rules

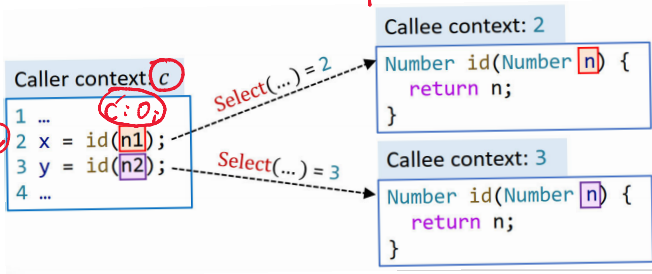
Kind	Statement	Rule (under context c)
New	`i: x = new T()`	$\dfrac{}{c: o_i \in pt(c: x)}$
Assign	`x = y`	$\dfrac{c': o_i \in pt(c: y)}{c': o_i \in pt(c: x)}$
Store	`x.f = y`	$\dfrac{c': o_i \in pt(c: x), c'': o_j \in pt(c: y)}{c'': o_j \in pt(c': o_i.f)}$
Load	`y = x.f`	$\dfrac{c': o_i \in pt(c: x), c'': o_j \in pt(c': o_i.f)}{c'': o_j \in pt(c: y)}$

Kind	Statement	Rule (under context c)
Call	`l: r = x.k(a1,…,an)`	$\dfrac{\begin{array}{c} c': o_i \in pt(c: x), \\ m = \text{Dispatch}(o_i, k), c^t = \text{Select}(c, l, c': o_i) \\ c'': o_u \in pt(c: aj), 1 \le j \le n \\ \Rightarrow c''': o_v \in pt(c^t: m_{ret}) \end{array}}{\begin{array}{c} c': o_i \in pt(c^t: m_{this}) \\ c'': o_u \in pt(c^t: m_{pj}), 1 \le j \le n \\ \Rightarrow c''': o_v \in pt(c: r) \end{array}}$

- Dispatch(o_i, k): resolves the virtual dispatch of k on o_i to a target method (based on type of o_i)
- Select$(c, l, c': o_i)$: selects context for target method m, based on the information available at call site l
- $c^t: m_{this}$: this variable of $c^t: m$
- $c^t: m_{pj}$: the j-th parameter of $c^t: m$
- ➢ $c^t: m_{ret}$: the variable that holds the return value of $c^t: m$

① 根据 receive object 的类型 (o_i 的 type of) + 方法签名 k 即可确定目标方法 m，不需等 x 具体指向对象 o_i 的上下文 c'.

Select函数则是在调用时根据[c, l, c:o_i]调用点的信息算出此调用所处上下文标识



②③④ 指向信息流动时，c:x的指向信息 c:o_i 只会传递给目标方法对应上下文的this指针 c^t:mthis，参数与返回值信息也只会在调用点 ⟷ 目标方法的对应上下文之间传递。（井水不犯河水）

上层法

- Recall context-insensitive pointer analysis

1. Build pointer flow graph (PFG)

Mutually dependent

2. Propagate points-to information on PFG

- Context-sensitive pointer analysis

带有上下文信息的CFG

1. Build pointer flow graph (PFG with C.S.)

Mutually dependent

2. Propagate points-to information on PFG with C.S.

构建PFG + 在PFG上传播指向信息

Solve(m^{entry})
WL=[ ], PFG={ }, S={ }, RM={ }, CG={ }
AddReachable([ ]: m^{entry})
while WL is not empty **do**
  remove $\langle n, pts \rangle$ from WL
  $\Delta = pts - pt(n)$
  Propagate(n, Δ)
  if n represents a variable $c: x$ **then**
    foreach $c': o_i \in \Delta$ **do**
      foreach `x.f = y` $\in S$ **do**
        AddEdge($c: y, c': o_i.f$)
      foreach `y = x.f` $\in S$ **do**
        AddEdge($c': o_i.f, c: y$)
      ProcessCall($c: x, c': o_i$)

AddReachable($c: m$)
  if $c: m \notin RM$ **then**
    add $c: m$ to RM
    $S \cup = S_m$
    foreach `i: x = new T()` $\in S_m$ **do**
      add $\langle c: x, \{c: o_i\} \rangle$ to WL
    foreach `x = y` $\in S_m$ **do**
      AddEdge($c: y, c: x$)

加入已知世界

ProcessCall($c: x, c': o_i$)
  foreach `l: r = x.k(a1,…,an)` $\in S$ **do**
    ① m = Dispatch(o_i, k)
    c^t = Select($c, l, c': o_i$)
    ② add $\langle c^t: m_{this}, \{c': o_i\} \rangle$ to WL
    if $c: l \rightarrow c^t: m \notin CG$ **then**
      add $c: l \rightarrow c^t: m$ to CG
      AddReachable($c^t: m$)
    ③ **foreach** parameter p_i of m **do**
      AddEdge($c: a_i, c^t: p_i$)
    ④ AddEdge($c^t: m_{ret}, c: r$)

S	Set of reachable statements
S_m	Set of statements in method m
RM	Set of **C.S.** reachable methods
CG	**C.S.** call graph edges

关于确定 c^t 的 Select 函数

- Call-site sensitivity
- Object sensitivity
- Type sensitivity
- ……

Select($c, l, c': o_i$)

caller context	call site	receiver object, with heap context
c	2	$c: o_1$

Caller context: c

```
1 C x = new T();
2 r = x.foo(a1, a2);
```

Callee context: c^t

```
class T … {
  B foo(A p1, A p2) {
    this…
    return ret;
  }}
```

Tian Tan @ Nanjing University

Ps: C.I. 就可以被视为 Select(_,_,_) = [ ] 的特殊情况。

1. Call-Site Sensitivity

- Each context consists of a list of call sites (call chain)
  - At a method call, append the call site to the caller context as callee context
  - Essentially the abstraction of call stacks

$$\text{Select}(\boldsymbol{c}, \boldsymbol{l}, \_) = [l', \dots, l'', l]$$
where $c = [l', \dots, l'']$

Also called call-string sensitivity, or k-CFA

* Olin Shivers, 1991. "*Control-Flow Analysis of Higher-Order Languages*". Ph.D. Dissertation. Carnegie Mellon University.

为了①确保 PTA 的终止（无穷递归一无穷上下文）

②防止调用链太长搞炸 PTA。

⇒ 设置一个调用链长度的上界 k（取后 k 个）

Ps：实际应用中，k 一般是一个比较小的数（$k \leq 3$），而且方法上下文与堆对象的上下文可以取不同的 k 长度。

2. Object Sensitivity

本质上也是使用调用路径抽象上下文，不过构成调用的不再是 call-site，而是发起调用的 Receive object $\begin{cases} c' \\ o_i \end{cases}$ ⇒ 唯一确定一个 Receive object

- Each context consists of a list of abstract objects (represented by their allocation sites)
  - At a method call, use the receiver object with its heap context as callee context
  - Distinguish the operations of data flow on different objects

$$\text{Select}(\_, \_, c' : \boldsymbol{o_i}) = [o_j, \dots, o_k, o_i]$$
$$\text{where } c' = [o_j, \dots, o_k]$$

Essentially "allocation-site sensitivity"

* Ana Milanova, Atanas Rountev, and Barbara G. Ryder. "*Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java*". ISSTA 2002.

对比 Call-site Sensitive & Object Sensitive

```
1  a1 = new A(); a1 → o1
2  a2 = new A(); a2 → o2
3  b1 = new B(); b1 → o3
4  b2 = new B(); b2 → o4
5  a1.set(b1);
6  a2.set(b2);
7  x = a1.get();
8
9  class A {
10   B f;
11   void set(B b) {
12     this.doSet(b);
13   }
14   void doSet(B p) {
15     this.f = p;
16   }
17   B get() {
18     return this.f;
19   }
20 }
```

$c^t = [5]$ (at line 7)

$c^t = [6]$ (at line 10)

$c^t = [12]$ (at line 13)

1-call-site

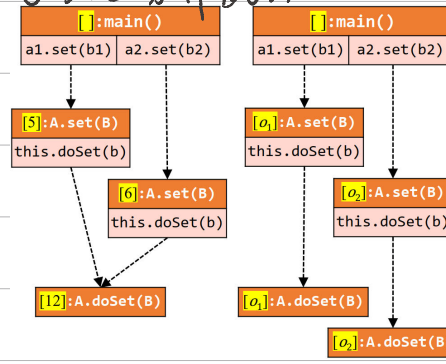
Variable	Object
$[5]: set_{this}$	o_1
$[5]: b$	o_3
$[6]: set_{this}$	o_2
$[6]: b$	o_4
$[12]: doSet_{this}$	o_1, o_2
$[12]: p$	o_3, o_4
Field	**Object**
$o_1.f$	o_3, o_4
$o_2.f$	o_3, o_4

1-object

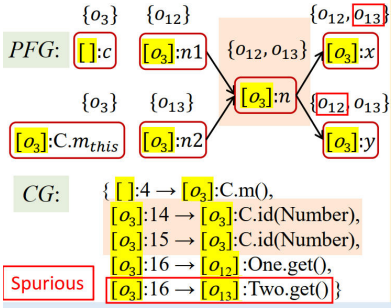
Variable	Object
$[o_1]: set_{this}$	o_1
$[o_1]: b$	o_3
$[o_1]: doSet_{this}$	o_1
$[o_1]: p$	o_3
$[o_2]: set_{this}$	o_2
$[o_2]: b$	o_4
$[o_2]: doSet_{this}$	o_2
$[o_2]: p$	o_4
x	o_3
Field	**Object**
$o_1.f$	o_3
$o_2.f$	o_4

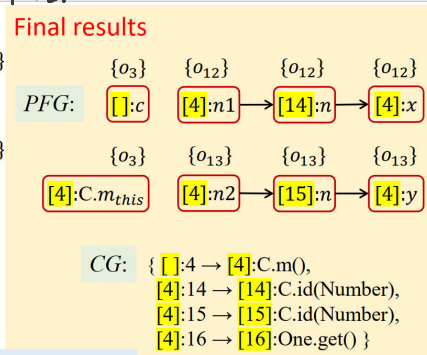
对于上面含有递归调用的情形，1-call-site 由于十分有限的调用路径记忆能力，往往会没有办法区分不同调用的上下文（实际上就是不同调用路径的后缀部分相同，而截取的长度 k

有限)。而使用 R_0 作为调用路径的节点，其实际上包含的"信息"是包含整条历史路径的(只是可能粒度较粗)，因此上述情况 Object Sen. 是很有可能可以区分开的。



但是有些情况下 Object S. 的调用路径会发生归并，而 Call-site S. 则不会：






```
1 class C {                    10   void m() {
2   static void main() {       11     Number n1,n2,x,y;
3     C c = new C();           12     n1 = new One();
4     c.m();                   13     n2 = new Two();
5   }                          14     x = this.id(n1);
6                              15     y = this.id(n2);
7   Number id(Number n) {      16     x.get();
8     return n;                17   }
9   }                          18 }
```

R_0 相同，都是 this 指向的对象，因此 14/15 时 id 的调用将无法区分，数据流会归并

理论上来说，没有办法直接比较两种方法对精度的影响，但对于OO语言来说，Object Sen. 一般表现更好。

	Time (s)		#may-fail-cast		#call-graph-edge	
	2-call	2-obj	2-call	2-obj	2-call	2-obj
batik	6,886	3,300	2,452	1,606	94,211	76,807
checkstyle	2,277	2,003	863	581	54,171	48,809
sunflow	5,570	1,208	2,504	1,837	100,701	89,866
findbugs	3,812	2,661	2,056	1,409	72,118	65,836
jpc	3,343	559	1,855	1,392	89,677	81,030
eclipse	1,896	146	886	546	42,872	38,151
chart	2,705	282	1,481	883	59,691	52,374
fop	5,503	1,200	1,975	1,446	79,524	71,408
xalan	1,927	1,093	919	533	48,763	44,871
bloat	5,712	3,525	1,699	1,193	58,696	53,143

For all numbers, lower is better (in terms of eff

In general
- Precision: object > call-site
- Efficiency: object > call-site

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. "*A Principled Approach to Selective Context Sensitivity for Pointer Analysis*". TOPLAS 2020.

3. Type Sensitivity

Object Sen. 的精度或许还不错，但速度有些慢，我们可以稍微放松一些对调用路径节点的约束，不再使用具体的对象，而是对象的"类型"。这里的类型不是具体的类型，而是此对象的 allocation-site 所属的类的类型。

也即不再区分在同一个类下分配的对象。

- Each context consists of a list of types
  - At a method call, use the type containing the allocation site of the receiver object with its heap context as callee context
  - A coarser abstraction over object sensitivity

$$\text{Select}(\_, \_, \boldsymbol{c'}\!: \boldsymbol{o_i}) = [t', \dots, t'', \text{InType}(o_i)]$$
$$\text{where } c' = [t', \dots, t'']$$

```
1 class X {
2   void m() {
3     Y y = new Y();
4   }
5 }
```

$\text{InType}(o_3) = \text{X}$ (**not** Y!)

其相较于 Object Sen. 放弃了一些精度，但提高了速度（在k相同的情况下严格成立）

	Time (s)			#may-fail-cast			#call-graph-edge		
	2-call	2-obj	2-type	2-call	2-obj	2-type	2-call	2-obj	2-type
batik	6,886	3,300	378	2,452	1,606	1,938	94,211	76,807	77,337
checkstyle	2,277	2,003	125	863	581	695	54,171	48,809	49,274
sunflow	5,570	1,208	197	2,504	1,837	2,247	100,701	89,866	90,967
findbugs	3,812	2,661	265	2,056	1,409	1,683	72,118	65,836	66,443
jpc	3,343	559	128	1,855	1,392	1,599	89,677	81,030	81,527
eclipse	1,896	146	57	886	546	665	42,872	38,151	38,337
chart	2,705	282	84	1,481	883	1,155	59,691	52,374	52,965
fop	5,503	1,200	251	1,975	1,446	1,753	79,524	71,408	71,847
xalan	1,927	1,093	99	919	533	729	48,763	44,871	45,444
bloat	5,712	3,525	74	1,699	1,193	1,486	58,696	53,143	54,279

For all numbers, lower is better (in terms

In general
- Precision: object > type > call-site
- Efficiency: type > object > call-site

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. "*A Principled Approach to Selective Context Sensitivity for Pointer Analysis*". TOPLAS 2020.