Parameterized Object Sensitivity for Points-to Analysis for Java

Authors: Ana Milanova, Atanas Rountev, Barbara G. Ryder

Presenter: Zheng Song

Outlines

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Outlines

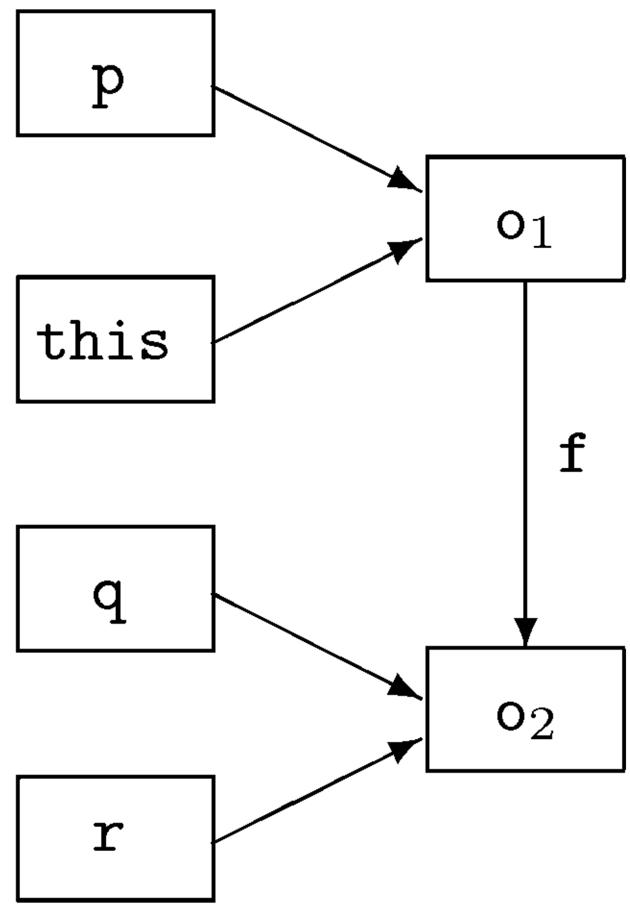
- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Introduction

- One sentence to conclude this paper: analyze a method separately for each of the objects on which this method is invoked
- For: Points-to Analysis: Method in Java to determine the set of objects pointed to by a reference variable or a reference object field

Sample points-to graph

```
class Y {...}
class X {
    Y f;
    void set(Y r)
         { this.f = r; }
    static void main() {
      s1:  X p = new X();
      s2:  Y q = new Y();
           p.set(q);
    }
}
```



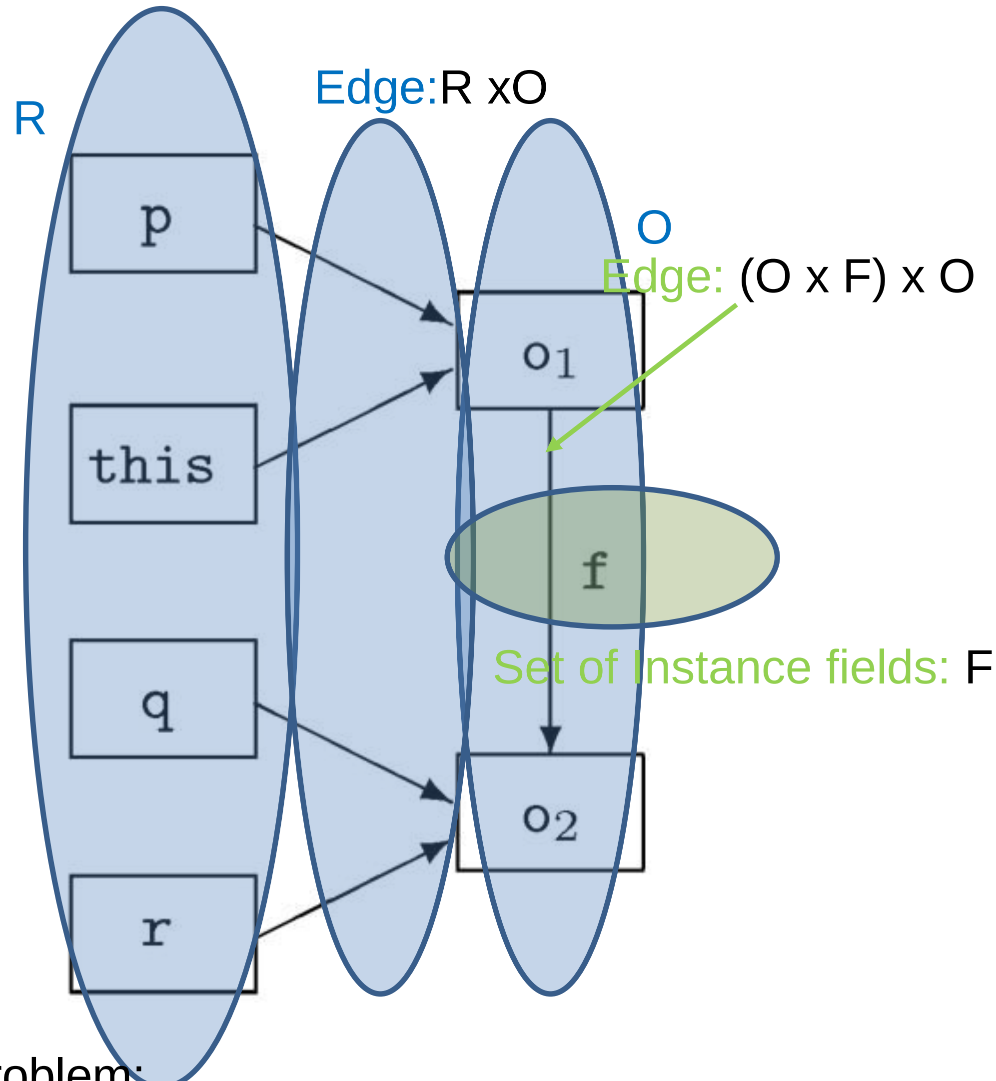
Outlines

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Existing Methods

- Andersen’s algorithm: flow insensitive & context insensitive
- Semantics (why called semantics?)
  - R – set of all reference variables
  - O – set of all objects created at object allocation sites
  - F – contains all instance fields in program class
  - Edge $(r,o_i) \in R \times O$
  - $(\langle o_i,f \rangle, o_j) \in (O \times F) \times O$
  - Transfer functions

Example



All statements are divided into :

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: `l =` r_0`.m(`r_1`,...,`r_k`)`

Go through each statement and conduct the graph following:

$$f(G, s_i : l = new\ C) = G \cup \{(l, o_i)\}$$

$$f(G, l = r) = G \cup \{(l, o_i) \mid o_i \in Pt(G, r)\}$$

$$f(G, l.f = r) = G \cup \{(\langle o_i, f\rangle, o_j) \mid o_i \in Pt(G, l) \ \wedge \ o_j \in Pt(G, r)\}$$

$$f(G, l = r.f) = G \cup \{(l, o_i) \mid o_j \in Pt(G, r) \ \wedge \ o_i \in Pt(G, \langle o_j, f\rangle)\}$$

$$f(G, l = r_0.m(r_1, \ldots, r_n)) = G \cup \{resolve(G, m, o_i, r_1, \ldots, r_n, l) \mid o_i \in Pt(G, r_0)\}$$

$$resolve(G, m, o_i, r_1, \ldots, r_n, l) = \texttt{let } m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_i, m) \texttt{ in } \{(p_0, o_i)\} \cup f(G, p_1 = r_1) \cup \ldots \cup f(G, l = ret_j)$$

Problem:

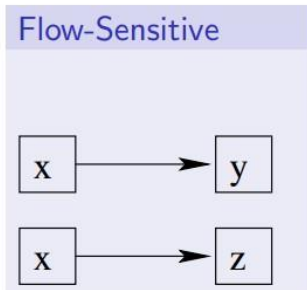
1. what’s the difference between OOPSLA’01?

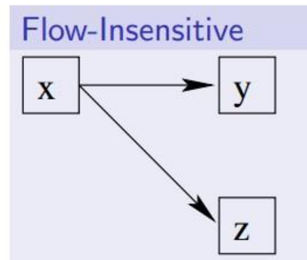
Existing Methods

- Flow insensitive V.S. Flow sensitive:

Code

```
int x;
int *y, *z;
x = &y;

x = &z;
```





Flow-sensitive analysis
- Computes one answer for every program point
- Requires iterative data-flow analysis or similar techniqu

Flow-insensitive analysis
- Ignores control flow
- Computes one answer for every procedure
- Can compute in linear time
- Less accurate than flow-sensitive

Is x constant?

```
void f(int x)
{
    x = 4;
    . . .
    x = 5;
}
```

Flow-sensitive analysis
- Computes an answer at every program point:
  - **x** is 4 after the first assignment
  - **x** is 5 after the second assignment

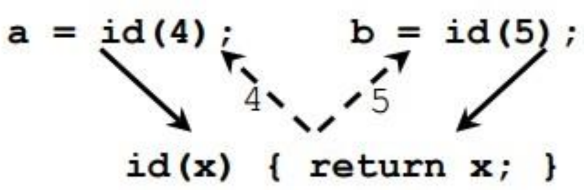
Flow-insensitive analysis
- Computes one answer for the entire procedure:
  - **x** is not constant

Existing Methods

- Context insensitive V.S. Context sensitive:

Context Sensitivity Example

Is x constant?

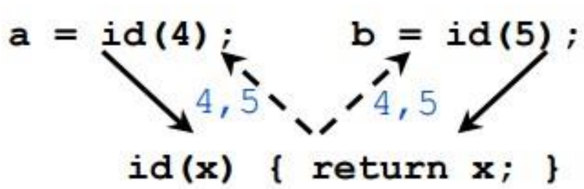


Context-sensitive analysis

- Computes an answer for every callsite:
  - **x** is 4 in the first call
  - **x** is 5 in the second call

Context-insensitive analysis

- Computes one answer for all callsites:
  - **x** is not constant
- Suffers from unrealizable paths:
  - Can mistakenly conclude that `id(4)` can return `5` because we merge (smear) information from all callsites



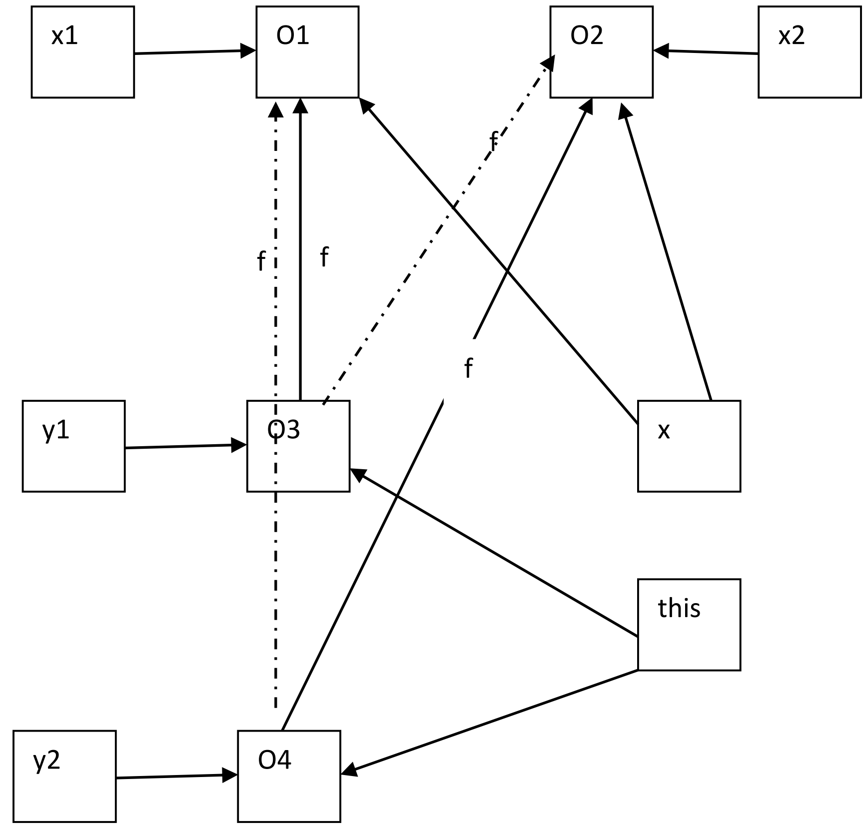
Its limitation in Object Oriented Programing

- Encapsulation
- Inheritance
- Collection (Containers)…

Lets try to analyze these features using flow insensitive and context insensitive analysis

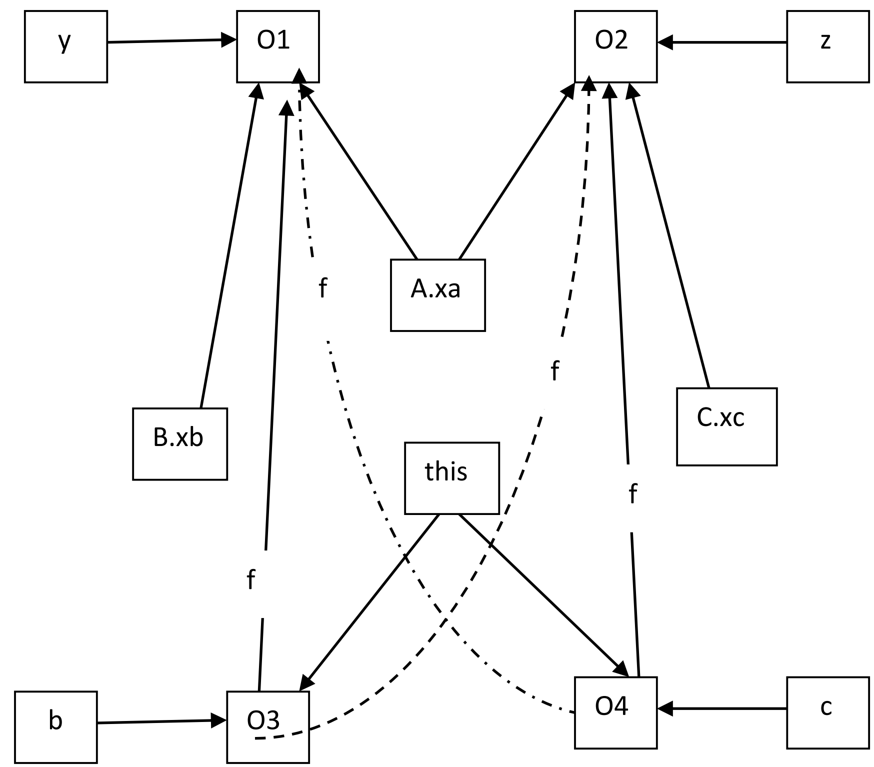
Encapsulation

```
class X {...}
class Y {
    X f;
    void set(X x) { this.f = x; } }
s1: X x1 = new X();
s2: X x2 = new X();
s3: Y y1 = new Y();
s4: Y y2 = new Y();
    y1.set(x1);
    y2.set(x2);
```



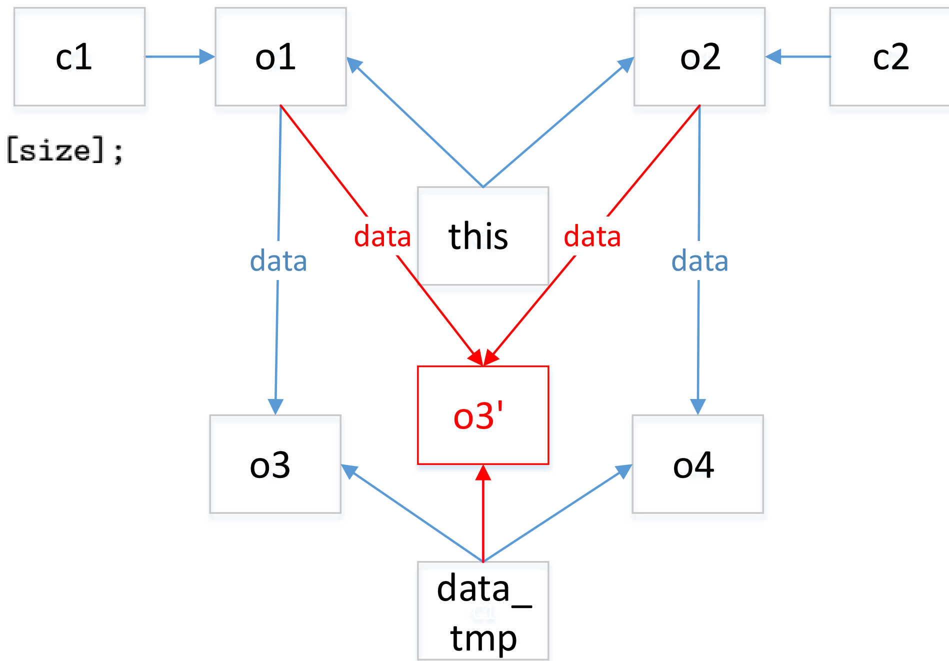
Inheritance

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }

   class A {
        X f;
1       A(X xa) { this.f = xa; } }

   class B extends A {
2       B(X xb) { super(xb); ...  }
        void m() {
3           X xb = this.f;
4           xb.n(); } }

   class C extends A {
5       C(X xc) { super(xc); ...  }
        void m() {
6           X xc = this.f;
7           xc.n(); } }

8   s1: Y y = new Y();
9   s2: Z z = new Z();
10  s3: B b = new B(y);
11  s4: C c = new C(z);
12      b.m();
13      c.m();
```



container

```
   class Container {
       Object[] data;
       Container(int size) {
1        s1: Object[] data_tmp = new Object[size];
2          this.data = data_tmp; }
       void put(Object e,int at) {
           Object[] data_tmp = this.data;
3          data_tmp[at] = e; }
       Object get(int at) {
4          Object[] data_tmp = this.data;
5          return data_tmp[at]; } }

6  s2: Container c1 = new Container(100);
7  s3: Container c2 = new Container(200);
8  s4: X x = new X();
9      c1.put(x,0);
10 s5: X y = new Y();
11     c2.put(y,1);
```



Imprecision

- Encapsulation
- Inheritance
- Container
  - are all strong concepts of OOP
  - But not captured properly with old techniques
  - Solution is Object sensitivity

Outlines

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Object Sensitivity

- With object sensitivity, each instance method and each constructor is analyzed separately for each object on which this method/constructor may be invoked.
- How? Revised semantics
  - O` - set of all object names
  - R` - set of replicas of reference variable
  - Relation α(C,m)=> α(o,m): C, or D which is a superclass of C
  - Set of new transfer functions

Object Sensitivity

Object Names

$O_{ij\ldots pq}$: the sequence of allocation sites (*si*, *sj* , . . . , *sp*, *sq*).

A particular name *oij···pq* represents all run-time objects that were created by *sq* when the enclosing instance method or constructor was invoked on an object represented by name *oij...p* which was created at allocation site *sp* (recursive)

S1: object O1

S2: object O2

S3: object O3

$O_1 \Rightarrow O_{21}$ & O_{31}

```
      class Container {
          Object[] data;
          Container(int size) {
1          s1: Object[] data_tmp = new Object[size];
2              this.data = data_tmp; }
          void put(Object e,int at) {
               Object[] data_tmp = this.data;
3              data_tmp[at] = e; }
          Object get(int at) {
4              Object[] data_tmp = this.data;
5              return data_tmp[at]; } }

6     s2: Container c1 = new Container(100);
7     s3: Container c2 = new Container(200);
8     s4: X x = new X();
9         c1.put(x,0);
10    s5: X y = new Y();
11        c2.put(y,1);
```

Object Sensitivity

Context Sensitivity

With more objects, next step we make more references pointing to these objects:

$$\mathcal{C} = O' \cup \{\epsilon\}$$
$$R \times \mathcal{C} \rightarrow R'.$$

[\epsilon is to deal with static calls]

If *r* is a local variable or a formal parameter of an instance method or a constructor *m*, the pair (*r*, *o*) is mapped to a "fresh" variable r_o for every context $o \in O'$ for which $\alpha(o, m)$ holds.

Fig. 4

$\alpha(o_3, \texttt{A.A})$ $\alpha(o_4, \texttt{A.A})$, two copies of A.this corresponding to contexts *o*3 and *o*4

Fig. 5

$\alpha(o_2, \texttt{Container.put})$ and $\alpha(o_3, \texttt{Container.put})$; therefore there are context copies of put.this, put.data tmp, and put.e corresponding to contexts *o*2 and *o*3

Q: how to formalize each element in R' ?

Breaking News…

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Parameterized Object Sensitivity

The formal definition of O' is as follows:

—$o_q \in O'$ for each $s_q \in S$ located in a static method

—if $o_{ij\ldots p} \in O'$ and $s_q \in S$ is located in an instance method or a constructor m such that $\alpha(o_{ij\ldots p}, m)$, then

(1) if $|ij \cdots p| < k$, then $o_{ij\cdots pq} \in O'$

(2) if $|ij \cdots p| = k$, then $o_{j\cdots pq} \in O'$

Notes:

1. The parameterized framework only apply to the set of objects (O'), by affecting (O'), it further affects R' and the transfer functions;
2. If k=1, it is actually Andersen's algorithm
3. K can be different to different statements

Q: why we need k?

Object Sensitivity

- *Transfer Functions*

$$F(G, s_q : l = new\ C) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(l^c, c \oplus_k s_q)\}$$

$$F(G, l = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c)$$

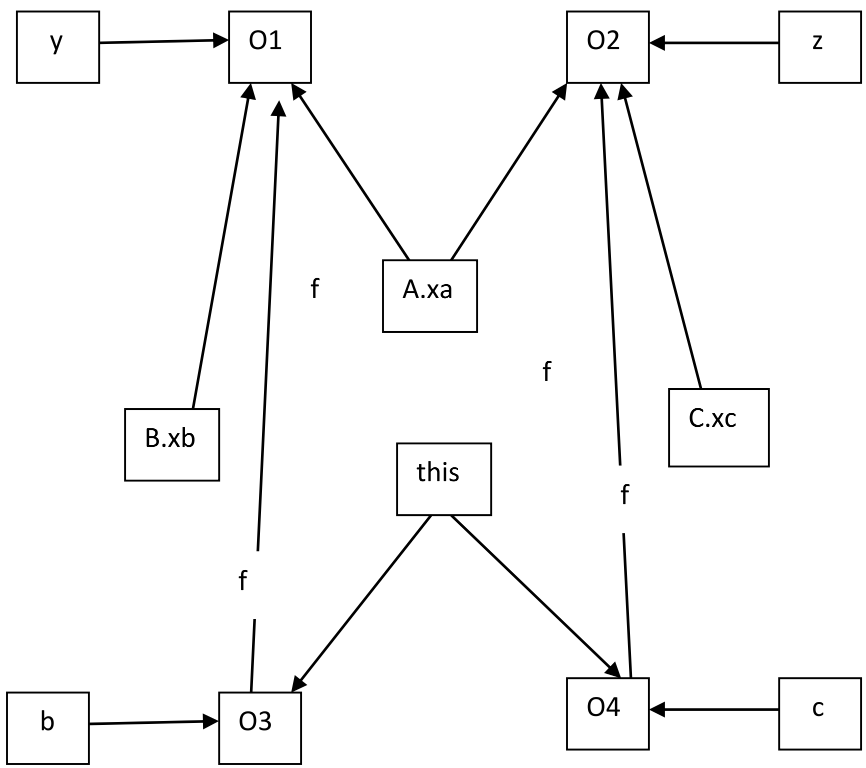
$$F(G, l.f = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c.f = r^c)$$

$$F(G, l = r.f) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c.f)$$

$$F(G, l = r_0.m(r_1, \ldots, r_n)) = G \cup \bigcup_{c \in \mathcal{C}_m} \{resolve(G, m, o, r_1^c, \ldots, r_n^c, l^c) \mid o \in Pt(G, r_0^c)\}$$

$$\begin{array}{l} resolve(G, m, o, r_1^c, \ldots, r_n^c, l^c) = \\ \quad \texttt{let}\ \ c' = o \\ \qquad m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o, m)\ \ \texttt{in} \\ \qquad\quad \{(p_0^{c'}, o)\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'}) \end{array}$$

Context sensitivity included



- B.this^{o3},B.xb^{o3}, A.xa^{o3}

C.this^{o4},C.xc^{o4}, A.xa^{o4}

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }

   class A {
       X f;
1      A(X xa) { this.f = xa; } }

   class B extends A {
2      B(X xb) { super(xb); ...  }
       void m() {
3          X xb = this.f;
4          xb.n(); } }

   class C extends A {
5      C(X xc) { super(xc); ...  }
       void m() {
6          X xc = this.f;
7          xc.n(); } }

8  s1: Y y = new Y();
9  s2: Z z = new Z();
10 s3: B b = new B(y);
11 s4: C c = new C(z);
12     b.m();
13     c.m();
```

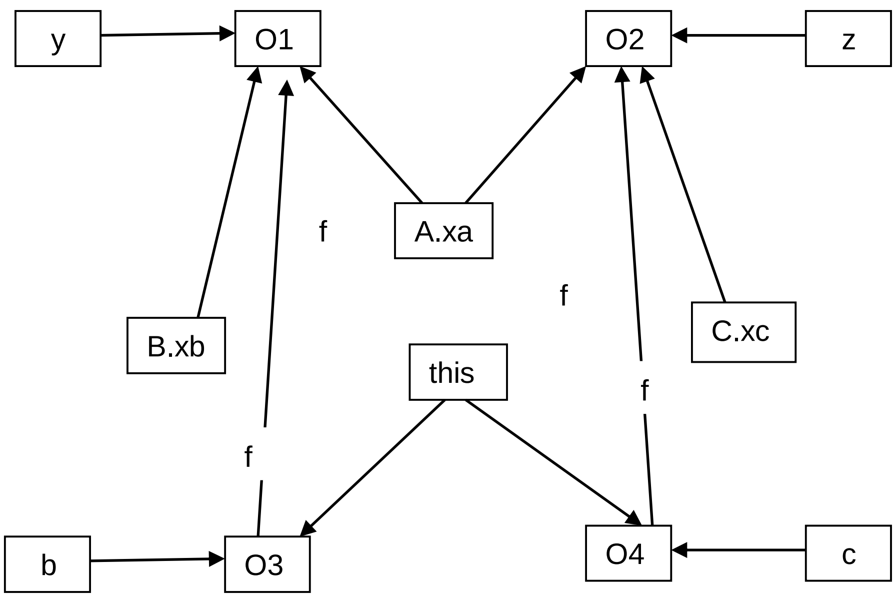
Example3.1.4

3.1.4 *Example.* Consider the set of statements in Figure 4. Since $\alpha(\texttt{B}, \texttt{B.B})$ and $\alpha(\texttt{B}, \texttt{A.A})$, we have

$$\{\texttt{B.this}^{o_3}, \texttt{B.xb}^{o_3}, \texttt{A.this}^{o_3}, \texttt{A.xa}^{o_3}\} \subseteq R'$$

Similarly, we have

$$\{\texttt{C.this}^{o_4}, \texttt{C.xc}^{o_4}, \texttt{A.this}^{o_4}, \texttt{A.xa}^{o_4}\} \subseteq R'$$



At line 2, $\texttt{B.this}^{o_3}$ points to o_3 and $\texttt{B.xb}^{o_3}$ points to o_1. When the analysis processes the call to `A.A` at line 2, `A.this` and `A.xa` are mapped to the context copies corresponding to o_3, and points-to edges $(\texttt{A.this}^{o_3}, o_3)$ and $(\texttt{A.xa}^{o_3}, o_1)$ are added to the graph. Similarly because of line 5, $\texttt{A.this}^{o_4}$ points to o_4 and $\texttt{A.xa}^{o_4}$ points to o_2. Statement `this.f=xa` at line 1 occurs in the context of o_3 and o_4. Thus,

$$\texttt{A.this}^{o_3} = \texttt{A.xa}^{o_3} \qquad \texttt{A.this}^{o_4} = \texttt{A.xa}^{o_4}$$

which produces edges $(\langle o_3, f\rangle, o_1)$ and $(\langle o_4, f\rangle, o_2)$. Since $\alpha(\texttt{B}, \texttt{B.m})$ and $\alpha(\texttt{C}, \texttt{C.m})$, we have

$$\{\texttt{B.m.this}^{o_3}, \texttt{B.m.xb}^{o_3}, \texttt{C.m.this}^{o_4}, \texttt{C.m.xc}^{o_4}\} \subseteq R'$$

When the analysis processes the statement at line 3, `B.m.xb` and `B.m.this` will be mapped to the context copies corresponding to o_3. Since $\texttt{B.m.this}^{o_3}$ points to o_3 and $\langle o_3, f\rangle$ points only to o_1, the statement at line 3 produces edge $(\texttt{B.m.xb}, o_1)$. Similarly, the statement at line 6 produces edge $(\texttt{C.m.xc}, o_2)$.

Advantages

- Models OOP features
- Distinguishes between different receiver objects
- Static methods and variables can be handled with insensitivity
- Can be parameterized

Outlines

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Side-effect Analysis (MOD)

input Stmt: set of statements $\quad \mathit{map}: R \times \mathcal{C} \rightarrow R'$

$\mathit{Methods}$: set of methods $\quad \mathit{Pt}: R' \rightarrow \mathcal{P}(O')$

output $\mathit{Mod}: \mathit{Stmt} \times \mathcal{C} \rightarrow \mathcal{P}(O')$

initialized to $\emptyset$ for all $(s, c) \in \mathit{Stmt} \times \mathcal{C}$

declare $\mathit{MMod}: \mathit{Methods} \times \mathcal{C} \rightarrow \mathcal{P}(O')$

initialized to $\emptyset$ for all $(m, c) \in \mathit{Methods} \times \mathcal{C}$

Instance field assignments:

- **foreach** instance field write $s: p.f = q \in \mathit{Stmt}$ **do**
  - **foreach** context c in which s appears **do**
    - $\mathit{Mod}(s, c) := \mathit{Pt}(\mathit{map}(p, c))$
    - add $\mathit{Mod}(s, c)$ to $\mathit{MMod}(\mathit{EnclMethod}(s), c)$

Virtual method calls:

- **while** changes occur in Mod or MMod **do**
  - **foreach** virtual call $s: l = r.m(\ldots) \in \mathit{Stmt}$ **do**
    - **foreach** context c in which s appears **do**
      - **foreach** object $o \in \mathit{Pt}(\mathit{map}(r, c))$ **do**
        - add $\mathit{MMod}(\mathit{target}(o, m), o)$ to $\mathit{Mod}(s, c)$ ← Typo: should be c
      - add $\mathit{Mod}(s, c)$ to $\mathit{MMod}(\mathit{EnclMethod}(s), c)$

Static method calls:

  - **foreach** static call $s: l = C.m(\ldots) \in \mathit{Stmt}$ **do**
    - **foreach** context c in which s appears **do**
      - $\mathit{Mod}(s, c) := \mathit{Mod}(s, c) \cup \mathit{MMod}(m, \epsilon)$
      - add $\mathit{Mod}(s, c)$ to $\mathit{MMod}(\mathit{EnclMethod}(s), c)$

Outlines

- Introduction
- Existing Method and its limitation
- Object Sensitive analysis
- Parameterized Object Sensitivity
- Implementation
- Evaluation
- Questions

Implementations

- Parameterized object-sensitive points-to analysis (context depth = 1):
  - *ObjSens1*: keeps context-sensitive information for implicit parameters `this` and formal parameters of instance methods and constructors.
  - *ObjSens2*: the same as *ObjSens1*, but it also keeps track of return variables.

Implementations

- Context-sensitive analysis based on the call string approach to context sensitivity, for a call string k = 1 (*CallSite*).
- Distinguishes context per call site.
- To allow for comparison, the context replication is performed for `this`, formal parameters and return variables in instance methods and constructors.

Characteristics of Programs

Table I. Characteristics of the Data Programs. First Two Columns Show the Number and Bytecode Size of User Classes. Last Three Columns Include Library Classes

Program	User Class	Size (Kb)	Whole-program Class	Whole-program Method	Whole-program Stmt
proxy	18	56.6	565	3283	58837
compress	22	76.7	568	3316	60010
db	14	70.7	565	3339	60747
jb-6.1	21	55.6	574	3393	60898
echo	17	66.7	577	3544	62646
raytrace	35	115.9	582	3451	62755
mtrt	35	115.9	582	3451	62760
jtar-1.21	64	185.2	618	3583	65112
jlex-1.2.5	25	95.1	578	3381	65437
javacup-0.10	33	127.3	581	3564	66463
rabbit-2	52	157.4	615	3770	68277
jack	67	191.5	613	3573	69249
jflex-1.2.2	54	198.2	608	3692	71198
jess	160	454.2	715	3973	71207
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Analysis Cost

Table II. Running Time and Memory Usage of the Analyses

Program	*And* Time [sec]	*And* Mem [Mb]	*CallSite* Time [sec]	*CallSite* Mem [Mb]	*ObjSens1* Time [sec]	*ObjSens1* Mem [Mb]	*ObjSens2* Time [sec]	*ObjSens2* Mem [Mb]
proxy	4.4	40	6.9	39	5.9	40	6.1	40
compress	12.0	46	12.0	47	8.8	46	13.2	46
db	12.2	47	11.4	47	11.7	48	12.2	46
jb	7.5	43	7.3	43	7.1	43	7.0	43
echo	27.3	60	24.8	59	24.3	59	27.2	59
raytrace	13.9	50	12.6	51	13.6	50	13.8	50
mtrt	15.4	50	12.9	51	15.2	50	15.6	50
jtar	23.3	58	19.9	56	21.4	56	20.6	56
jlex	8.5	46	9.0	46	8.8	46	8.7	46
javacup	13.1	57	17.3	56	16.8	53	15.1	56
rabbit	16.1	52	14.5	52	13.9	52	13.9	52
jack	38.4	62	38.1	62	37.5	62	37.6	62
jflex	20.2	71	20.1	70	19.7	70	19.7	70
jess	24.6	67	26.1	67	24.3	66	26.9	67
mpegaudio	15.4	52	14.1	54	16.2	52	13.2	54
jjtree	12.4	53	11.8	53	11.2	53	8.7	51
sablecc	68.7	115	40.1	94	62.3	113	35.9	94
javac	427.6	121	430.7	123	430.2	120	416.9	120
creature	85.2	100	61.1	86	55.7	88	57.6	86
mindterm	44.5	91	44.9	89	49.9	88	42.6	92
soot	80.8	130	92.6	132	69.8	128	89.7	132
muffin	110.0	144	108.4	135	99.6	132	101.1	133
javacc	76.2	112	82.3	112	81.4	116	80.1	112

MOD Analysis Precision

Table III. Number of Modified Objects for Program Statements. Each Column Shows the Percentage of Statements Whose Number of Modified Objects is in the Corresponding Range

	And			*CallSite*			*ObjSens2*		
Program	1–3	4–9	≥10	1–3	4–9	≥10	1–3	4–9	≥10
proxy	19%	6%	75%	25%	7%	68%	75%	14%	11%
compress	23%	4%	73%	27%	5%	67%	67%	9%	24%
db	20%	4%	76%	24%	4%	72%	48%	25%	27%
jb	15%	5%	80%	20%	5%	75%	67%	20%	13%
echo	25%	6%	69%	30%	5%	65%	63%	11%	26%
raytrace	23%	5%	72%	28%	6%	66%	66%	9%	25%
mtrt	23%	5%	72%	28%	6%	66%	66%	9%	25%
jtar	18%	8%	74%	24%	7%	69%	61%	15%	24%
jlex	17%	4%	79%	20%	4%	76%	56%	34%	10%
javacup	14%	3%	83%	21%	4%	75%	53%	38%	9%
rabbit	18%	5%	77%	23%	6%	71%	47%	13%	40%
jack	17%	3%	80%	20%	3%	77%	53%	8%	39%
jflex	19%	4%	77%	23%	5%	72%	54%	34%	12%
jess	15%	5%	80%	25%	3%	72%	60%	9%	31%
mpegaudio	23%	4%	73%	28%	4%	68%	65%	9%	26%
jjtree	8%	2%	90%	10%	2%	88%	32%	26%	42%
sablecc	20%	3%	77%	32%	4%	64%	52%	15%	33%
javac	14%	4%	82%	18%	6%	76%	37%	5%	58%
creature	18%	3%	79%	27%	3%	70%	54%	13%	33%
mindterm	20%	8%	73%	25%	7%	68%	55%	16%	29%
soot	16%	4%	80%	25%	8%	67%	43%	15%	42%
muffin	16%	4%	80%	24%	4%	72%	45%	7%	48%
javacc	10%	1%	89%	11%	1%	88%	29%	49%	22%
Average	18%	4%	78%	23%	5%	72%	54%	18%	28%

Conclusions

- Presented a framework for parameterized object-sensitive points-to analysis, and side-effect and def-use analyses based on it.
- Object-sensitive analysis achieves significantly better precision than context-insensitive analysis, while remaining efficient and practical.

Acknowledgement

- Besides the original paper and its journal version, some materials are derived from
  - UPENN CIS570 Lecture Notes
  - UMD CMSC737(Fundamentals of Software Testing), Student Presentation by Anand Bahety& Dan Bucatanschi