

Practical Program Analysis of Object-oriented Software Part 1

Dr. Barbara G. Ryder
Rutgers University



<http://www.cs.rutgers.edu/~ryder>

September 2006

Outline

Part 1

- What is program analysis for OOPLs?
- Reference analyses
 - Type-based
 - Flow-based
 - Flow sensitivity
 - Field sensitivity
 - Context sensitivity
 - 1-CFA, Object-sensitive

Outline

Part 2

- **New results on accommodating reflection in points-to analysis**
- **Applications of analysis in software tools to increase programmer productivity**
 - Using infeasibility analysis to enable better test coverage for recovery code
 - Combining static and dynamic analysis to report change impact of edits

What is program analysis?

- **Static program analysis extracts information about program semantics from code, without running it**
 - E.g., Def-use analysis for dependences
 - Builds an abstract representation of the program and solves the analysis problem
- **Dynamic program analysis extracts information from a program execution**
 - E.g., Profiling, dynamic slicing
- **Our focus: static reference analyses**

Object-oriented PL

- Characterized by data abstraction, inheritance, polymorphism
- Allows dynamic binding of method calls
- Allows dynamic loading of classes
- Allows querying of program semantics at runtime through reflection

Reference Analysis

- *Determines information about the set of objects to which a reference variable or field may point during program execution*

Reference Analysis

- OOPLs need type information about objects to which reference variables can point to resolve dynamic dispatch
- Often data accesses are indirect to object fields through a reference, so that the set of objects that might be accessed depends on which object that reference can refer at execution time
- Need to pose this as a compile-time program analysis with representations for reference variables/fields, objects and classes.

Reference Analysis enables...

- Construction of possible calling structure of program
 - Dynamic dispatch of methods based on runtime type of receiver `x.f();`
- Understanding of possible dataflow in program
 - Indirect side effects through reference variables and fields `r.g=`

Uses of Reference Analysis Information in Software Tools

- **Program understanding tools (flow)**
 - Semantic browsers
 - Program slicers
- **Software maintenance tools (type, flow)**
 - Change impact analysis tools
- **Testing tools (flow)**
 - Coverage metrics

Example Analyses

- **Type hierarchy-based**
 - CHA, RTA
- **Incorporating flow**
 - FieldSens (Andersen-based points-to)
- **Incorporating flow and calling context**
 - 1-CFA
 - Object-sensitive

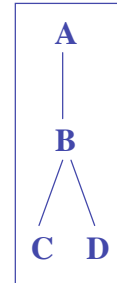
Example

```

static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
    
```

```

class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
    
```



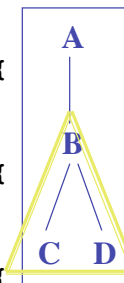
CHA Example

```

static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
    
```

```

class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
    
```



Cone(Declared_type(receiver))

CHA Characteristics

- Ignores program flow
- Calculates types that a reference variable can point to
- Uses 1 abstract reference variable per class throughout program
- Uses 1 abstract object to represent all possible instantiations of a class

J. Dean, D. Grove, C. Chambers, *Optimization of OO Programs Using Static Class Hierarchy*, ECOOP'95

cf Frank Tip, OOPSLA'00

RTA Example

```
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}

static void f(A a2){
    a2.foo();
}

static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}

class A {
    foo(){..}
}

class B extends A{
    foo() {...}
}

class C extends B{
    foo() {...}
}

class D extends B{
    foo() {...}
}
```

RTA Characteristics

- Only analyzes methods *reachable* from `main()`, on-the-fly
- Ignores classes which have not been instantiated as possible receiver types
- Uses 1 abstract reference variable per class throughout program
- Uses 1 abstract object to represent all possible instantiations of a class

D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls", OOPSLA'96

Clients of CHA & RTA

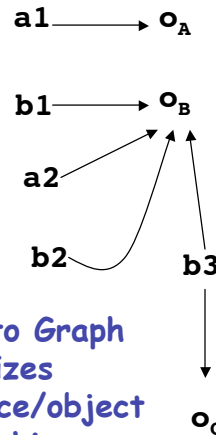
- Call graph construction
 - Estimate dynamic dispatch targets
 - A pre-requisite for all static analyses of OOPs that trace interprocedural flow
 - E.g., slicing, obtaining method coverage metrics for testing, understanding calling structure of legacy code, heap optimization, etc

FieldSens Example

```

static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}

```



Points-to Graph summarizes reference/object relationships

cf Frank Tip, OOPSLA'00

Intl School SE, Salerno, Sept'06, B & Ryder-1

17

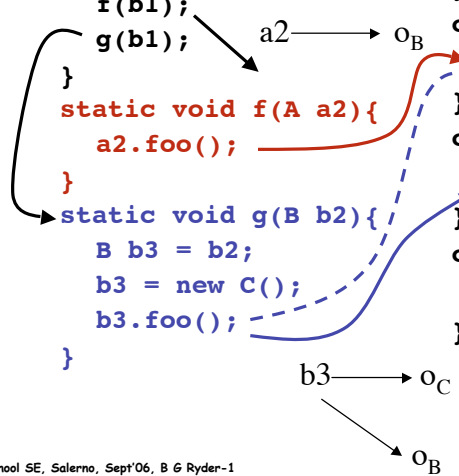
FieldSens Example

```

static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}

class A {
  foo(){..}
}
class B extends A{
  foo() {...}
}
class C extends B{
  foo() {...}
}
class D extends B{
  foo() {...}
}

```



cf Frank Tip, OOPSLA'00

Intl School SE, Salerno, Sept'06, B & Ryder-1

18

FieldSens Characteristics

- Only analyzes methods *reachable* from `main()`
 - On-the-fly call graph construction
- Keeps track of individual reference variables and fields
- Groups objects by their creation site
- Incorporates reference value flow in assignments and method calls

A. Rountev, A. Milanova, B. G. Ryder, "Points-to Analysis for Java Using Annotated Constraints". OOPSLA'01

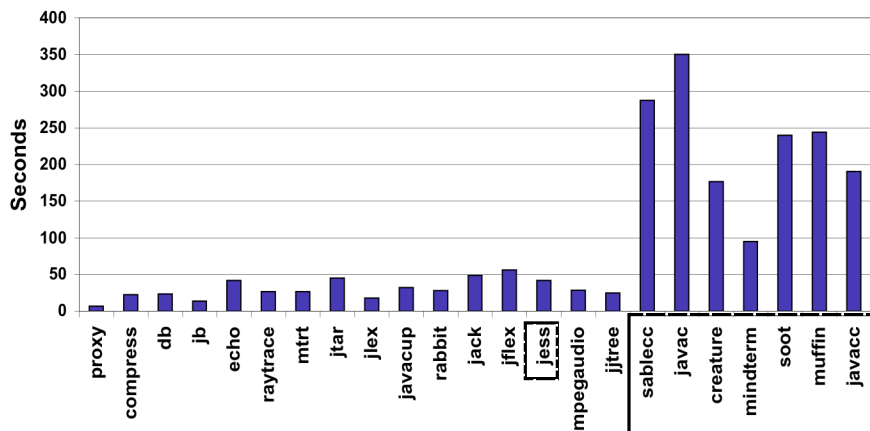
Some Clients of FieldSens

- Improving precision of call graphs with truly polymorphic call sites
- Calculating object read/write's through references
- Calculating objects escaping from their creation environment (e.g., a thread)

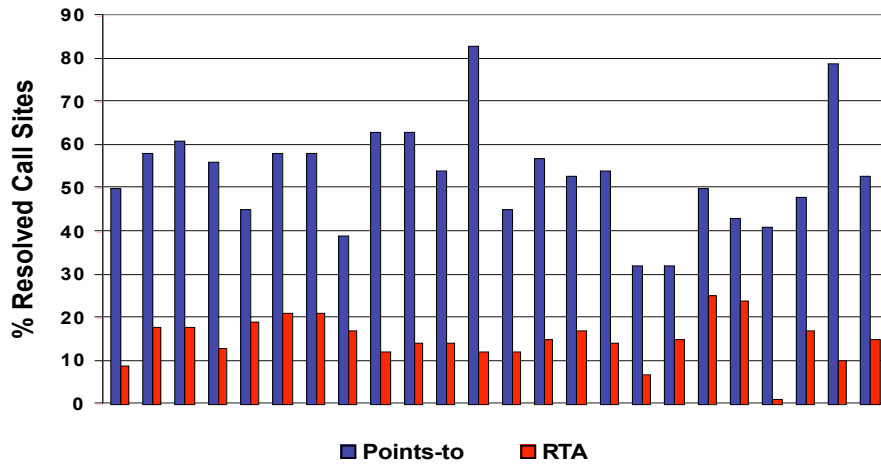
Experiments

- 23 Java programs: 14 - 677 user classes, with 57K-1,070K bytecode
 - Added the necessary library classes (JDK 1.1)
 - Machine: 360 MHz, 512Mb SUN Ultra-60
- Cost measured in time and memory
- Points-to algorithm presented as an inclusion constraint solution problem
 - Migrated Andersen's points-to analysis for C to Java
- Precision (wrt usage in client analyses and transformations)
 - Call graph construction
 - Thread-local heap discovery
 - Object read-write information

Analysis Time



Resolution of Virtual Call Sites

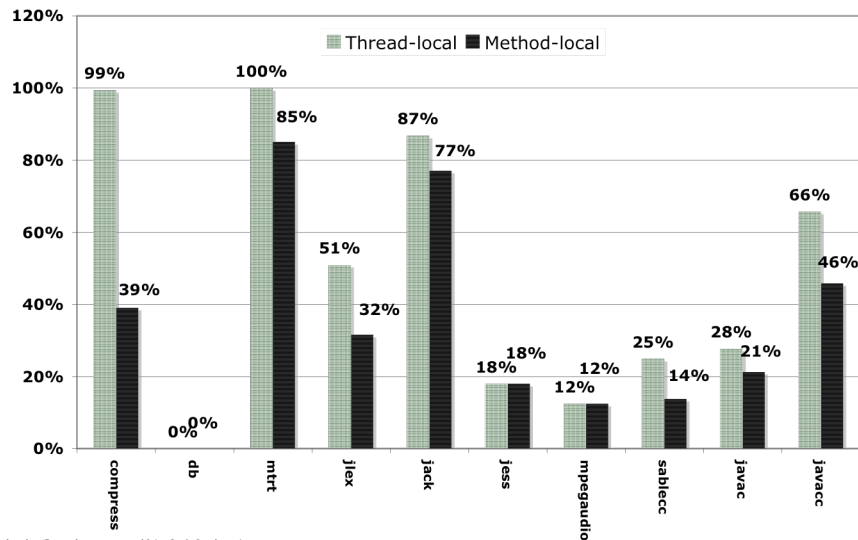


Call sites all reported as polymorphic by CHA

Number of Objects Created

Program	Objects Created
compress	456
db	154,325
mtrt	6,457,298
jlex	7,350
jack	1,340,919
jess	7,902,221
mpegaudio	2,025
sablecc	420,494
javac	3,738,777
javacc	43,265

Thread & Method-local new()'s



Intl School SE, Salerno, Sept'06, B & Ryder-1

25

Object Read-Write Info

- Measured number of objects accessed on average at indirect reads/writes
 - More than 1/2 the accesses were to a single object (the lower bound) and
 - On average 81% were resolved to at most 3 objects

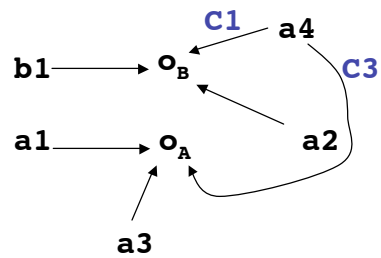
Intl School SE, Salerno, Sept'06, B & Ryder-1

26

1-CFA Analysis

Improves on FieldSens by keeping track of calling context

```
static void main(){  
    B b1 = new B(); //OB  
    A a1 = new A(); //OA  
    A a2, a3;  
    C1: a2 = f(b1);  
    C2: a2.foo();  
    C3: a3 = f(a1);  
    C4: a3.foo();  
}  
public static A f(A a4){return a4;}
```



Points-to Graph

at C2, main calls B.foo()
at C4, main calls A.foo()

Intl School SE, Salerno, Sept'06, B G Ryder-1

27

1-CFA Characteristics

- Only analyzes methods *reachable* from main()
- Keeps track of individual reference variables and fields
- Groups objects by their creation site
- Incorporates reference value flow in assignments and method calls
- Differentiates points-to relations for different calling contexts

Intl School SE, Salerno, Sept'06, B G Ryder-1

28

Dimensions of Precision

- Independent characteristics of a reference analysis which determines its precision
- Different combinations of these dimensions have already been explored in algorithms
- Need to understand what choices are available to design new analyses of appropriate precision for clients

Dimensions of Precision

- **Program representation - Call graph**
 - Use hierarchy-based approximation
 - Do reference analysis based on an already built approximate call graph - Palsberg'91, Chatterjee'99, Sundaresan'00, Liang'01
 - Lazy on-the-fly construction
 - Only explore methods which are statically reachable from the main() (especially library methods)
 - Interleave reference analysis and call graph construction - Oxhoj'92, Razafimahefa'99, Rountev'01, Grove'01, Liang'01, Milanova'02, Whaley'02

Dimensions of Precision

• Object Representation

- **Use one abstract object per class** -
Hierarchy-based analyses: Dean'95, Bacon'96; Flow-based analyses: Diwan'96, Palsberg'91, Sundaresan'00, Tip'00
- **Group object instantiations by creation site** - Points-to analyses: Grove'01, Liang'01, Rountev'01, Milanova'02, Whaley'02
- **Finer-grained object naming** - Oxhoj'92, Plevyak'94, Grove'01, Liang'02, Milanova'02

Dimensions of Precision

Field Sensitivity

- **Field-independent(fi)**
 - Do not distinguish reference fields of an object, - Rountev'01, Lhotek & Hednren CC'03
- **Field-based(fb)**
 - Use one abstract field per field name (across all objects), -Lhotek & Hendren, CC'03
- **Field-sensitive(fs)**
 - Use one abstract field per field per abstract object (usually a creation site), -Rountev'01, Lhotek & Hednren CC'03

Spark Experiments

- Precision measure incorporated unreachable dereferences and unique object reference targets
 - Precision of fb:fs was 57.7:60.0 on average
 - Time cost was very similar
 - Space cost of fb:fs was 86.6:138.4 on average
- Lesson learned: sometimes less precision is okay - need to know the client of the points-to info

Dimensions of Precision

- Reference representation
 - Use one abstract reference per class - Dean'95, Bacon'96, Sundaresan'00
 - Use one abstract reference for each class per method - Tip'00
 - Represent reference variables or fields by their names program-wide - Sundaresan'00, Liang'01, Rountev'01, Milanova'02
 - **XTA**: example of one abstract reference for each class per method

XTA Analysis

- Calculates set of classes that reach a method, incorporating (limited) flow
- Uses an on-the-fly constructed call graph
- Uses one abstract object per class with distinct fields
- Uses one abstract reference per class in each method

F. Tip and J. Palsberg, "Scalable Propagation-based Call Graph Construction Algorithms", OOPSLA'00

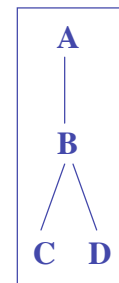
cf Frank Tip, OOPSLA'00

Example of XTA

{A,B}

```
static void main(){
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2){
    a2.foo();
}
static void g(B b2){
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
```

```
class A {
    foo(){..}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
```



Dimensions of Precision

• Directionality

- How flow in reference assignments ($r=s$) is interpreted by the analysis
 - Symmetric (Unification): r and s have same points-to set after the assignment - Ruf'00, Liang'01
 - Directional (Inclusion): r 's points-to set includes s 's points-to set after the assignment - Sundarasan'00, Rountev'01, Liang'01, Milanova'02, Whaley'02

Dimensions of Precision

• Flow sensitivity

- Analyses which capture the sequential order of execution of program statements
 - Diwan'96, Chatterjee'99, Whaley'02

- E.G.,

```
1. A s, t;  
2. s = new A(); //o1  
3. t = s;  
4. s = new A(); //o2
```

flow-sensitive:

at 2., s refers to o_1
at 3., s, t refer to o_1
at 4., s refers to o_2
 t refers to o_1

flow-insensitive:

s, t refer to $\{o_1, o_2\}$

Imprecision of Context Insensitivity

```
class Y extends X{ ... }
```

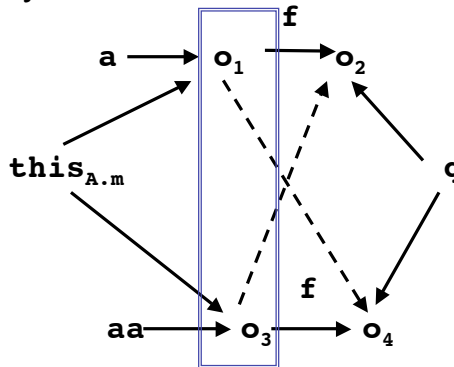
```
class A{  
  X f;  
  void m(X q)  
  { this.f=q; }  
}
```

```
A a = new A(); //o1
```

```
a.m(new X()); //o2
```

```
A aa = new A(); //o3
```

```
aa.m(new Y()); //o4
```



Dimensions of Precision

- **Context sensitivity**
 - Analyses which distinguish different calling contexts - Sharir/Pnueli'81
 - Call string - Palsberg'91, Grove'01
 - Functional approach- Plevyak'94, Agesen'95, Milanova'02
 - **1-CFA**, example of call string approach
 - **ObjSens**, example of functional approach

ObjSens Analysis

- Based on Andersen's points-to for C
- Uses receiver object to distinguish different calling contexts
- Groups objects by creation sites
- Represents reference variables and fields by name program-wide
- Flow-insensitive

A. Milanova, A. Rountev, B. G. Ryder, "Parameterized Object-sensitivity for Points-to and Side-effect Analyses for Java" ISSTA'02.

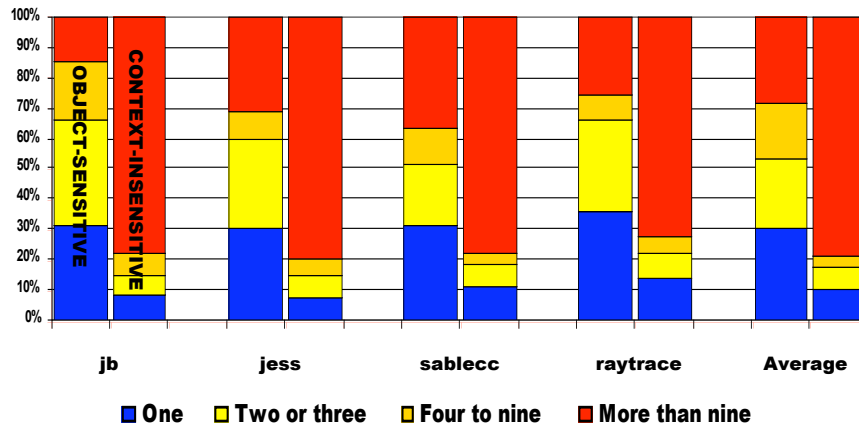
A. Milanova, A. Rountev, B.G. Ryder, "Parameterized Object Sensitivity for Points-to Analysis for Java", in *ACM Transactions on Software Engineering Methodology*, Volume 14, Number 1, pp 1-41, January 2005.

ObjSens Analysis

- Shown to analyze OO programming idioms well
 - Field encapsulation using set methods
`this.f=x`
 - Superclass constructor setting subclass object fields
 - Uses of containers

Side-effect Analysis: Modified Objects Per Statement

Milanova, ISSTA'02

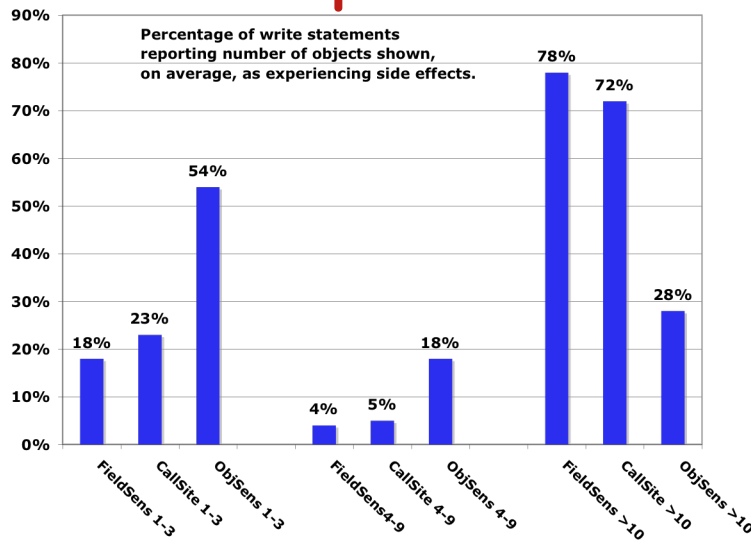


Intl School SE, Salerno, Sept'06, B & Ryder-1

43

Side Effect Analysis Comparison

Milanova, TOSEM05



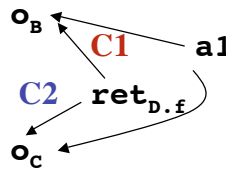
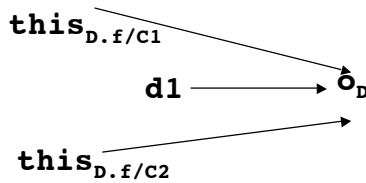
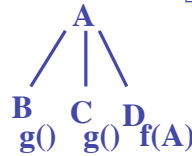
Intl School SE, Salerno, Sept'06, B & Ryder-1

44

1-CFA more precise than ObjSens

```
static void main(){
  D d1 = new D();
  if (...) C1: (d1.f(new B())).g();
  else C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```

1-CFA

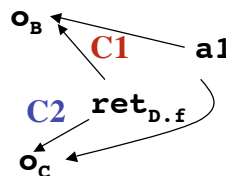
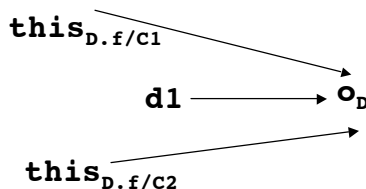


1-CFA more precise than ObjSens

```
static void main(){
  D d1 = new D();
  if (...) C1: (d1.f(new B())).g();
  else C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```

1-CFA

1-CFA distinguishes the two calling contexts of D.f at C1 and C2:
 At **C1**, B.g() called;
 At **C2**, C.g() called;



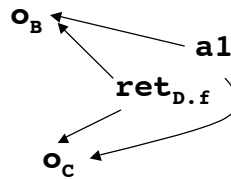
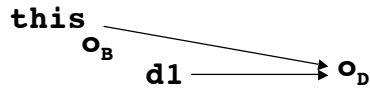
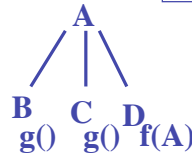
1-CFA more precise than ObjSens

```

static void main(){
  D d1 = new D();
  if (...) C1: (d1.f(new B())).g();
  else C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}

```

ObjSens



1-CFA more precise than ObjSens

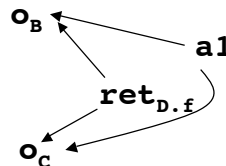
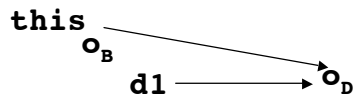
```

static void main(){
  D d1 = new D();
  if (...) C1: (d1.f(new B())).g();
  else C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}

```

ObjSens

ObjSens groups the two calling contexts of D.f with the same receiver at **C1** and **C2**; Both B.g(), C.g() are called at **C1** and **C2**;

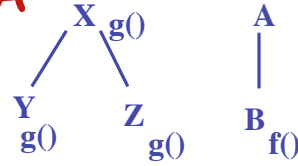


ObjSens more precise than 1-CFA

```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;

```

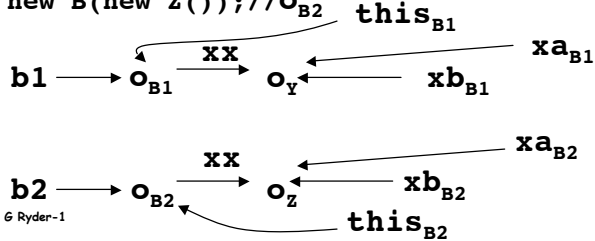


ObjSens

```

C1: B b1 = new B(new Y()); // OB1
C2: B b2 = new B(new Z()); // OB2

```

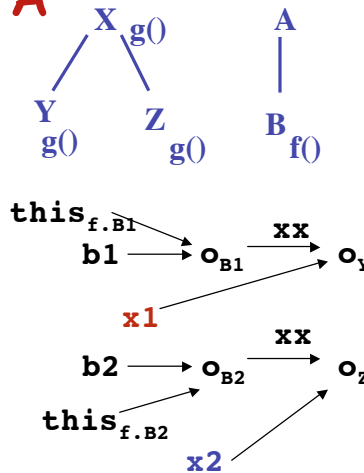


ObjSens more precise than 1-CFA

```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
    C1: B b1 = new B(new Y()); // OB1
    C2: B b2 = new B(new Z()); // OB2
    x1=b1.f();
    C4: x1.g();
    x2=b2.f();
    C5: x2.g();
  }

```



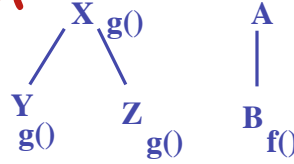
ObjSens

ObjSens more precise than 1-CFA

```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
    C1: B b1 = new B(new Y());//oB1
    C2: B b2 = new B(new Z());//oB2
    x1=b1.f();
    C4: x1.g();
    x2=b2.f();
    C5: x2.g();
  }
}

```



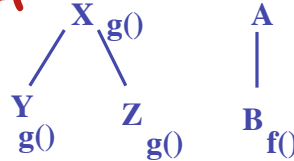
ObjSens finds
C4 calls Y.g() and
C5 calls Z.g()

ObjSens more precise than 1-CFA

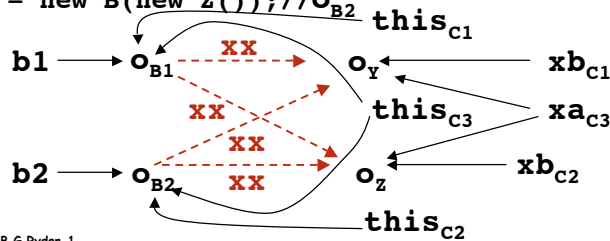
```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
    C1: B b1 = new B(new Y());//oB1
    C2: B b2 = new B(new Z());//oB2

```



1-CFA

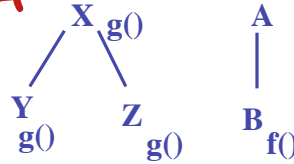


ObjSens more precise than 1-CFA

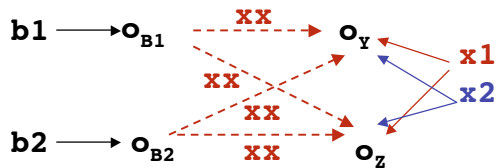
```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
    C1: B b1 = new B(new Y());//oB1
    C2: B b2 = new B(new Z());//oB2
    x1=b1.f();
    C4: x1.g();
    x2=b2.f();
    C5: x2.g();
  }
}

```



1-CFA

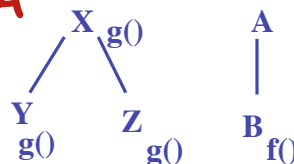


ObjSens more precise than 1-CFA

```

public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
    C1: B b1 = new B(new Y());//oB1
    C2: B b2 = new B(new Z());//oB2
    x1=b1.f();
    C4: x1.g();
    x2=b2.f();
    C5: x2.g();
  }
}

```



1-CFA finds
C4 calls Y.g(), Z.g() and
C5 calls Y.g(), Z.g()

Comparison Conclusion

- The call string and functional approaches to context sensitivity are incomparable!
- Neither is more powerful than the other
- Recent papers show that object-sensitive is effective in static analysis of race conditions (Aiken et. al, PLDI'06)

Difficult Issues

- Use of reflection and dynamic class loading
 - Need whole program for a safe analysis
- Java native methods
 - Need to model possible effects
- Exceptions
- Incomplete programs
- Lack of benchmarks