Advanced Program Analyses for Object-oriented Systems

Dr. Barbara G. Ryder
Rutgers University
http://www.cs.rutgers.edu/~ryder
http://prolangs.rutgers.edu/
July 2007

Lecture 2 - Outline

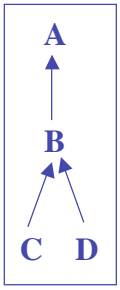
- Type-based call graph construction
- Dimensions of analysis precision
  - Properties and characterizations of an analysis
  - Choices and consequences (precision & cost)
- Reference analysis of OO programs
  - Points-to analyses
    - Flow sensitivity, context sensitivity, field sensitivity
  - Client analyses: Side effects, finding thread/method-local objects, synchronization removal, efficient heap layout

Running Example

cf Tip & Palsberg, OOPSLA'00

```
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(){…}
}
```

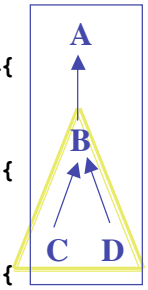


CHA Example

cf Tip & Palsberg, OOPSLA'00

```
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(){…}
}
```



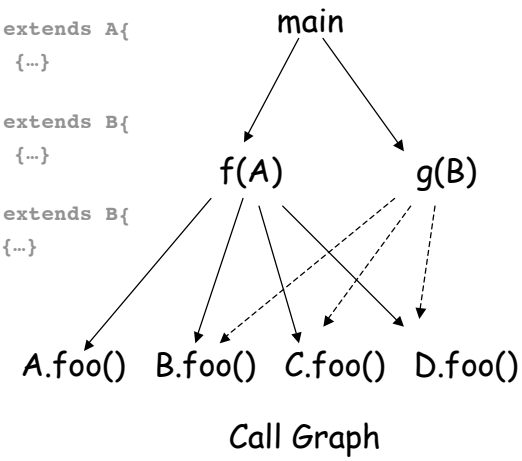
Cone(Declared_type(receiver))

CHA Example - Call Graph

cf Tip & Palsberg, OOPSLA'00

```
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(){…}
}
```



Call Graph

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

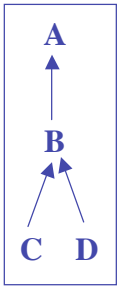
J. Dean, D. Grove, C. Chambers, *Optimization of OO Programs Using Static Class Hierarchy*, ECOOP'95

RTA Example

cf Tip & Palsberg, OOPSLA'00

```
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(){…}
}
```

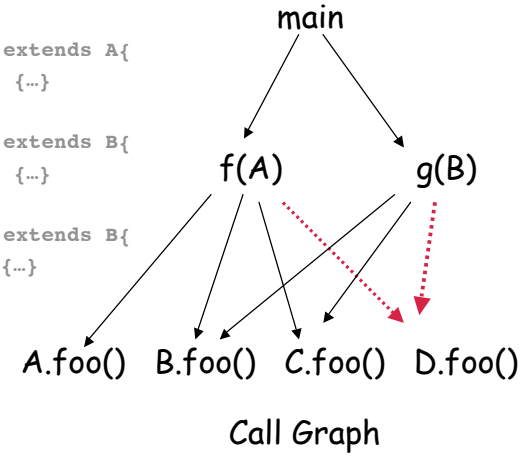


RTA Example

cf Tip & Palsberg, OOPSLA'00

```
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(){…}
}
```



Call Graph

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

Experimental Comparison C++ Programs

Bacon and Sweeney, OOPSLA'96

Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	11,157	IDL specification to C++ stub-code translator
lcom	17,278	Compiler for the "L" hardware description language
hotwire	5,335	Scriptable graphical presentation builder
simulate	6,672	Simula-like simulation class library and example
idl	30,288	SunSoft IDL compiler with demo back end
taldict	11,854	Taligent dictionary benchmark
deltablue	1,250	Incremental dataflow constraint solver
richards	606	Simple operating system simulator

Table 1: Benchmark Programs. Size is given in non-blank lines of code

Data Characteristics

Bacon and Sweeney, OOPSLA'96

- Frequency of execution matters
  - Direct calls were 51% of static call sites but only 39%of dynamic calls
  - Virtual calls were 21% of static call sites but were 36% of dynamic calls
- Results they saw differed from previous studies of C++ virtuals
  - Stresses importance of benchmarks in empirical work

Findings

Bacon and Sweeney, OOPSLA'96

- RTA was better than CHA on virtual function resolution, but not on reducing code size
  - Inference is that call graphs constructed have same node set but not same edge set!
- Claim both algorithms cost about the same because the dominant cost is traversing the CFGs of methods and identifying call sites
  - Implemented CHA with reachability calculation
- Claim that RTA is good enough for call graph construction so that more precise analyses are not necessary for this task

Type-based vs Flow-based Reference Analysis

- Uses only class hierarchy
- Same points-to set for every reference of a type
- Always flow- and context- insensitive
- Inexpensive
- Okay for call graph construction, but too imprecise for some other applications

- Uses reference assignments
- Distinguishes points-to sets of different references of same type
- Can be flow/context-sensitive or insensitive
- May be expensive
- Related to points-to approaches for C
- Okay for side-effect and dependence calculations

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

B.G. Ryder, "Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages", CC 2003, pp 126-137.

Dimensions of Precision

1. Program representation - Call graph
   - Use type-based approximation
   - Lazy, on-the-fly construction
     - Only explore methods which are statically reachable from the main()
     - Especially important for OOPLs use of libraries

Dimensions of Precision

2. Object Representation
   - Use one abstract object per class (CHA, RTA)
   - Group object instantiations by creation site
   - Finer-grained object naming
3. Field Sensitivity
   - May or may not distinguish fields of objects; field-sensitive, field-based, field-insensitive

Field Sensitivity

- Field-insensitive
  - Does not distinguish between fields of an abtract object
- Field-based
  - Collapses all same-named fields into an abstract representative
- Field-sensitive
  - Distinguishes between different fields of an abstract object

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

Lhotak and Hendren, "Scaling Java Points-to Analysis Using SPARK", CC'03

Dimensions of Precision

4. Reference representation

- Use one abstract reference per class (CHA, RTA)
- Use one abstract reference for each class per method (XTA)
- Represent reference variables or fields by their names program-wide (FieldSens)

Dimensions of Precision

5. Flow sensitivity

- Analyses which capture the sequential order of execution of program statements

```
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}

Dimensions of Precision

6. Context sensitivity

- Analyses which distinguish different calling contexts of the same method
- Differ by how they represent calling context
  - Call string
  - Functional approach
- 1-CFA, example of call string approach
- ObjSens, example of functional approach

Dimensions of Precision

7. 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
  - Directional (Inclusion): r's points-to set includes s's points-to set after the assignment

Reference Analyses

- Vary according to the choices taken in each of the dimensions
- Flow-based analyses based on ideas from points-to analysis of C pointers
- XTA: a flow-based analysis close to type-based analysis in cost, but with better precision because it incorporates flow of types into methods

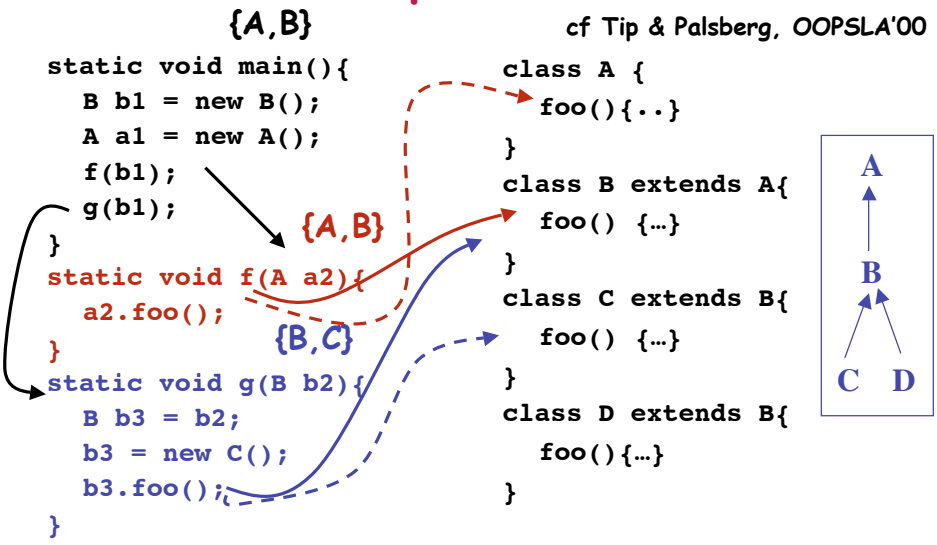
XTA Analysis

- 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 (field-sensitive)
- Uses one abstract reference per class in each method

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

Example of XTA

cf Tip & Palsberg, OOPSLA'00



Java Program Dataset

cf Tip & Palsberg, OOPSLA'00

benchmark	# classes	# methods	#fields (reference-typed)	# virtual call sites
Hanoi	44	379	232 (107)	285
Ice Browser	76	761	500 (253)	922
mBird	2,050	17,946	6739 (4284)	3,269
Cindy	468	4,449	3075 (1677)	5,085
CindyApplet	468	4,449	3075 (1677)	2,502
eSuite Sheet	588	5,590	4305 (1412)	4,459
eSuite Chart	733	8,302	5448 (2141)	8,074
javaFig 1.43	161	2,108	1526 (971)	3,482
BLOAT	282	2,677	1255 (541)	6,623
JAX 6.3	309	2,754	1252 (579)	3,836
javac	210	1,512	1107 (406)	3,621
Res. System	2,332	21,495	12487 (6334)	23,640

Findings

cf Tip & Palsberg, OOPSLA'00

- Measures precision improvements over RTA
  - Given that reference r can point to an RTA-calculated set of types program-wide, then XTA reduces the size of this set by 88%, on average, per method.
- The reachable methods set (i.e., call graph nodes) is minimally reduced over that of RTA
- The number of edges in the call graph is significantly reduced by XTA over RTA (.3%-29% fewer, 7% on average)

Findings, cont.

cf Tip & Palsberg, OOPSLA'00

- Data gives comparison restricted to those calls that RTA found to be polymorphic and how these analyses can improve on that finding.
  - Claim that the reduction in edges are for those calls that RTA found to be polymorphic, and often call sites become monomorphic after analysis
- Bottom line: Improved call graph construction

Data of Findings

cf Tip & Palsberg, OOPSLA'00

similar ⇩ ⇩ ⇩ ⇩ ⇩ better

benchmark	RTA unreached	RTA mono	RTA poly	XTA unreached	XTA mono	XTA poly
Hanoi	34.0%	61.6%	4.4%	34.0%	62.7%	3.3%
Ice Browser	4.0%	91.4%	4.7%	4.0%	91.6%	4.5%
mBird	14.2%	73.4%	12.3%	17.4%	70.9%	11.7%
Cindy	49.3%	45.0%	5.7%	49.4%	45.5%	5.0%
CindyApplet	72.0%	24.6%	3.4%	72.3%	24.5%	3.2%
eSuite Sheet	28.1%	68.4%	3.5%	28.2%	69.1%	2.8%
eSuite Chart	13.3%	76.6%	10.1%	15.7%	76.0%	8.3%
javaFig 1.43	9.1%	87.1%	3.9%	9.7%	87.2%	3.1%
BLOAT	6.6%	82.4%	11.1%	7.0%	82.2%	10.8%
JAX 6.3	18.7%	75.9%	5.4%	18.9%	76.8%	4.3%
javac	3.0%	77.6%	19.4%	3.0%	77.7%	19.3%
Res. System	18.1%	72.0%	9.9%	18.2%	74.0%	7.9%
AVERAGE			7.8%			7.0%

Fully Incorporating Flow

- Need to incorporate intraprocedural flow of data into and out of reference variables (i.e., assignments)
- Model parameter / argument associations as assignments
- FieldSens based on Andersen's points-to analysis for C
  - Flow-insensitive, context-insensitive, field-sensitive, inclusion constraints

Rules of Algorithm

- 4 types of reference assignment statements with points-to effects
  - Allocation: p=new X()
    - Adds o_X to Pts(p)
  - Copy: p = q
    - Pts(p) ⊇ Pts(q) (i.e., If o ∈Pts(q), then o ∈Pts(p))
  - Field store: p.f = q
    - If o∈Pts(p) and r ∈Pts(q), then r ∈ Pts(o.f)
  - Field load: p = q.g
    - If o ∈ Pts(q) and oo ∈ Pts(o.g) then oo ∈ Pts(p)

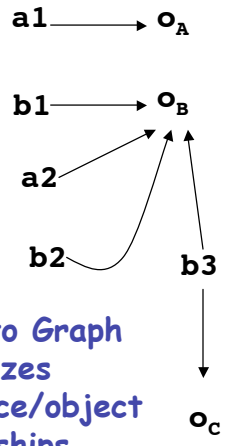
Points-to Graph

- Nodes represent reference variables and fields, or objects
- Edges represent possible run-time points-to relations
  - Sometimes labeled with object field names
- Uses: call graph construction, side effect analysis, memory optimizations

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();
}
```

cf Tip & Palsberg, OOPSLA'00



Points-to Graph summarizes reference/object relationships

cf Tip & Palsberg, OOPSLA'00

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();
}
```

a2 → o_B

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

b3 → o_C

o_B

FieldSens 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

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints", OOPSLA'00;
Lhotak and Hendren, "Scaling Java Points-to Analysis using SPARK", CC'03

Clients of FieldSens

Rountev et. al, OOPSLA'00

- Devirtualization
- Side effect analysis
  - What objects can have their values changed through a reference assignment?
- Memory optimizations
  - What objects can be stored on the stack frame (local to a method, or thread)?

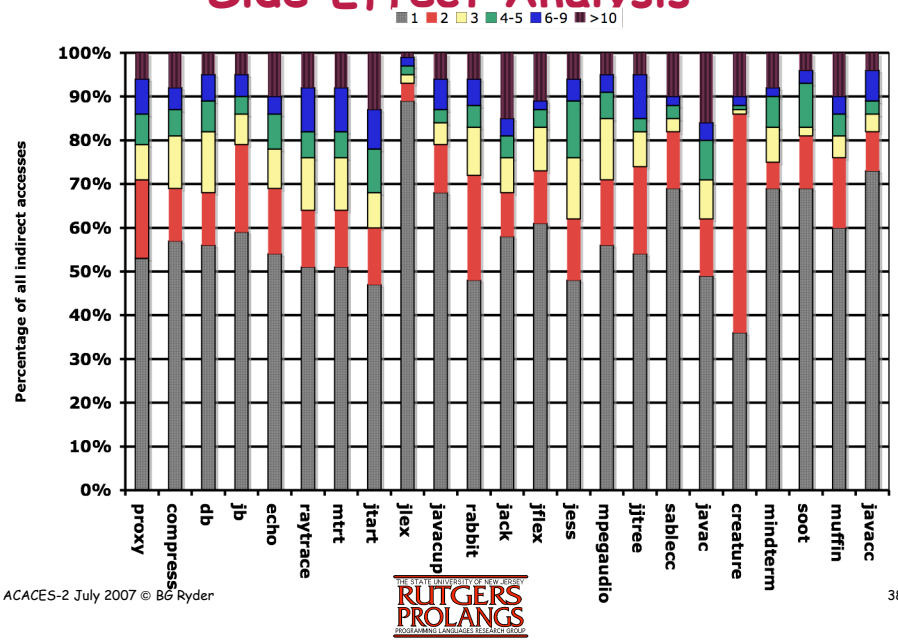
Benchmarks Used

Rountev et. al, OOPSLA'00

Program	User Class	Size (Kb)	Whole-program		
			Class	Method	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

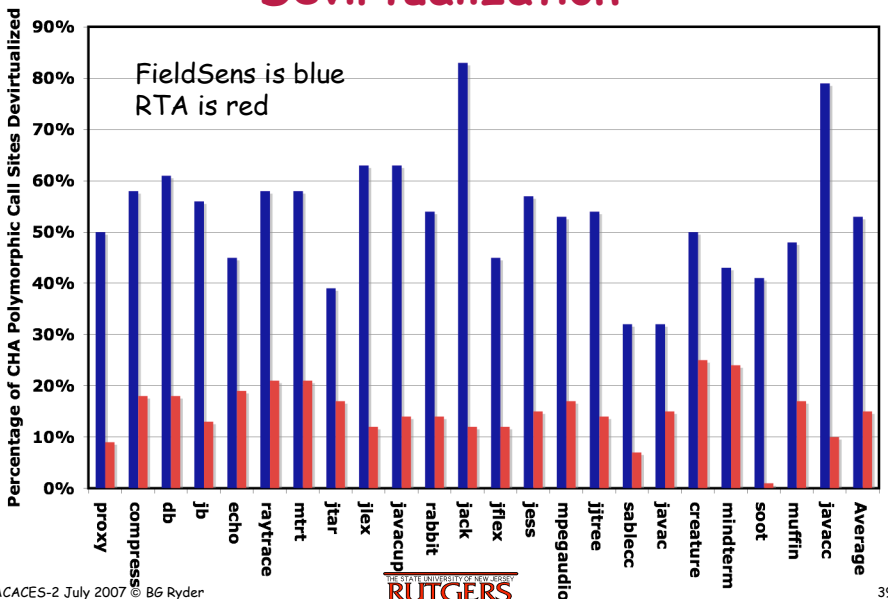
Rountev et. al, OOPSLA'00

Side Effect Analysis



Rountev et. al, OOPSLA'00

Devirtualization



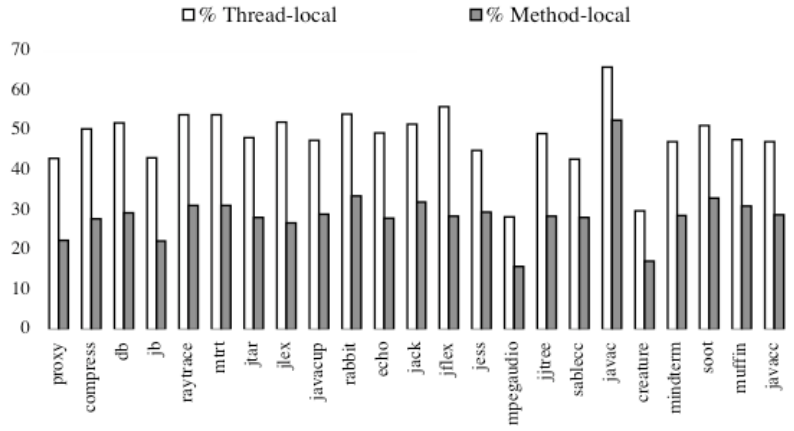
Synchronization Removal

- Java library methods are often synchronized for use in multi-threaded applications
- If program is single-threaded or threads do not share data, then this is unnecessary
  - Use *escape analysis* to find objects which escape the thread that creates them
    - If none found, then no need for synchronization
  - Can use results of points-to analysis to estimate objects that escape a method or thread

Thread-local & method-local *new* sites

Rountev et. al, OOPSLA'00

(a) Object allocation sites



Run-time Payoff

Rountev et. al, OOPSLA'00

(b) Run-time objects

Program	Objects	Thread-local	Method-local
compress	456	99.3%	39.0%
db	154325	0.03%	0.01%
mtrt	6457298	99.9%	85.0%
jlex	7350	50.9%	31.6%
jack	1340919	86.7%	77.0%
jess	7902221	17.9%	17.9%
mpegaudio	2025	12.4%	12.4%
sablecc	420494	24.9%	13.7%
javac	3738777	27.6%	21.2%
javacc	43265	65.7%	45.8%

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
```