# Mix Your Contexts Well

## Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity

Manas Thakur[1] and V. Krishna Nandivada[2]

CC 2020

[1]IIT Mandi, India [2]IIT Madras, India

Feb 23, 2020

# Context sensitivity

Your response should be sensitive to the context – Anonymous

- A popular way to improve the precision of program analysis, specially for OO programs.
- Compared to context-*insensitive* analyses:
  - Usually more **precise**
  - Usually **unscalable**

# Context sensitivity

Your response should be sensitive to the context – Anonymous

- A popular way to improve the precision of program analysis, specially for OO programs.
- Compared to context-*insensitive* analyses:
  - Usually more **precise**
  - Usually **unscalable**
- A method may be analyzed multiple times
  - Once in each unique context from which it may be called.


# Many context abstractions

"What is right" depends on the context – Upanisheds

- Several popular context abstractions in literature:
  - Call-site sensitivity [Sharir & Pnueli 1978]
  - Object-sensitivity [Milanova et al. 2005]
  - Value contexts [Khedker & Karkare 2008]
  - LSRV contexts [Thakur & Nandivada 2019]
  - All above with heap cloning [Nystrom et al. 2004]
- The choice of *context abstraction* plays an important role in determining the precision and scalability of the analysis.
- Relative advantages in terms of precision/scalability not well established.


# Call-site sensitivity

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     A a,b,c,d;
       ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  } }
```

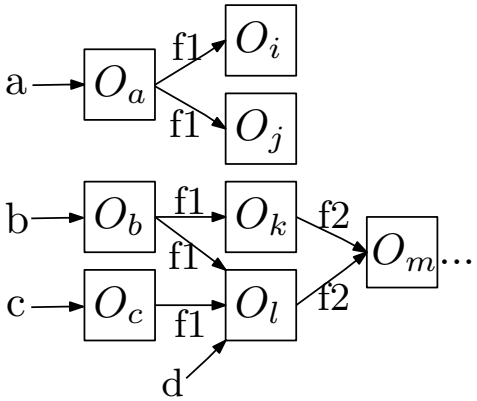
```
// Assume fb doesn't
// access caller's heap
```

- 2 contexts for bar
  - foo_5
  - foo_6
- 4 contexts for fb
  - foo_5+bar_11
  - foo_5+bar_12
  - foo_6+bar_11
  - foo_6+bar_12
- In case of recursion?

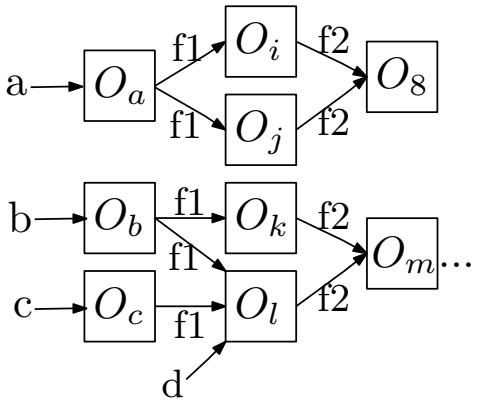

# Value contexts [CC'08]

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     A a,b,c,d;
       ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  } }

 // Assume fb doesn't
 // access caller's heap
```

**Points-to graph**

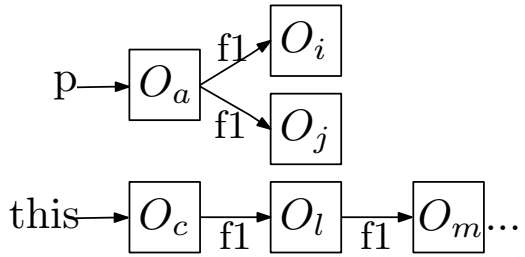



(Line 5)

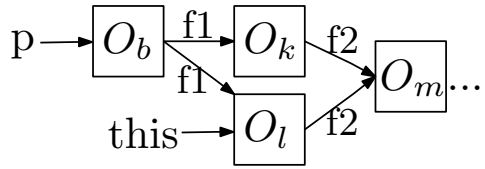



(Line 6)

**Value-context**




(Line 5)




(Line 6)

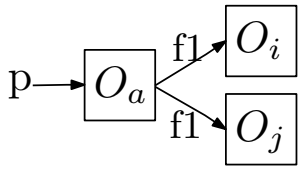
bar: analyzed twice, but fb: only once; generally scales better.

# Recent advance: LSRV contexts [CC'19]

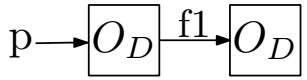
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     A a,b,c,d;
       ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  } }

 // Assume fb doesn't
 // access caller's heap
```

**(Level Summarized Relevant Value Contexts)**

Line 5:

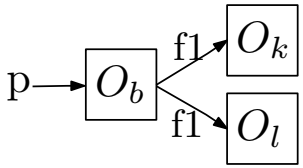



Relevant value-context

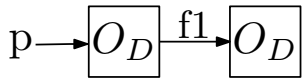



LSRV context

Line 6:




Relevant value-context



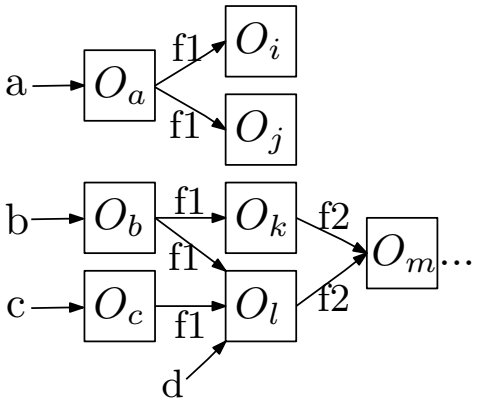

LSRV context

**Result:** `bar` and `fb` both analyzed only once!

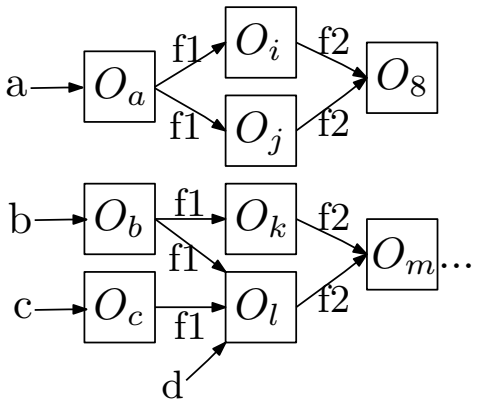
# Another popular choice: Object sensitivity

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     A a,b,c,d;
       ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  } }
```

**Points-to graph**




(Line 5)




(Line 6)

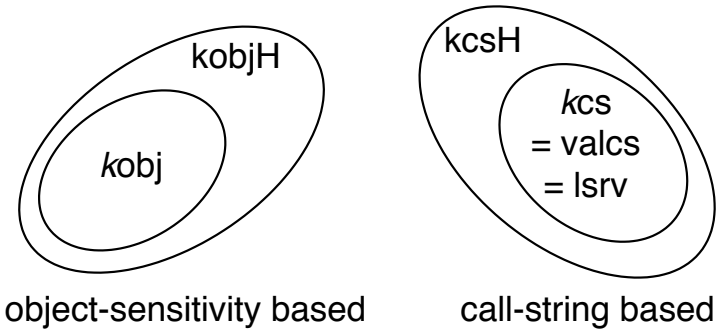
**Object-sensitivity**:
2 contexts for bar:

Line 5: Receiver $O_c$

Line 6: Receiver $O_l$

# What we know

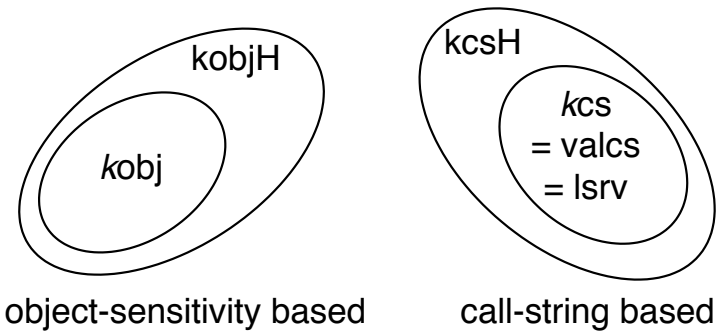
What I know is limited, what I don't is unlimited! – Common folklore.




- $k$-call-site-sensitive, value contexts and LSRV contexts have the same precision [CC'08, CC'19].
- Adding heap cloning improves the precision of call-site- as well as object-sensitivity.
- The relative precisions of object-sensitivity and call-site/value-contextx/LSRV-contexts are incomparable.

# What we know

What I know is limited, what I don't is unlimited! – Common folklore.




- $k$-call-site-sensitive, value contexts and LSRV contexts have the same precision [CC'08, CC'19].
- Adding heap cloning improves the precision of call-site- as well as object-sensitivity.
- The relative precisions of object-sensitivity and call-site/value-contextx/LSRV-contexts are incomparable.

**Our Goal**: Get the best of both worlds.

# Adding heap cloning

**Heap cloning:**

- Specializes objects (allocation sites) with the context in which they are created.
  - Object allocated at line $l$ in context $c$ represented as $O_{l\_c}$.
- Improves the partitioning efficacy of context-sensitivity.
- Usually generates more optimization opportunities, but with an increased analysis cost.

# Surprise #1 with heap cloning: *kcsH* vs *valcsH*

Expectations reduce the joy, surprise enhances joy! - Anonymous

```
1 class D {
2   void m1() {
3     P p = new P();
4     Q q1 = p.m2();
5     Q q2 = p.m2(); }
6   void m2() {
7     return new Q();
8   } /*m2*/
9 } /*class D*/
```

- Recall: $valcs \equiv_{precision} kcs$


# Surprise #1 with heap cloning: *kcsH* vs *valcsH*

Expectations reduce the joy, surprise enhances joy! - Anonymous

```
1 class D {
2   void m1() {
3     P p = new P();
4     Q q1 = p.m2();
5     Q q2 = p.m2(); }
6   void m2() {
7     return new Q();
8   } /*m2*/
9 } /*class D*/
```

- Recall: $valcs \equiv_{precision} kcs$
- *kcs*: m2 is analyzed twice.
- *valcs*: m2 is analyzed once.
- Both report that q1 and q2 are aliases after line 5.

# Surprise #1 with heap cloning: *kcsH* vs *valcsH*

Expectations reduce the joy, surprise enhances joy! - Anonymous

```
1 class D {
2   void m1() {
3     P p = new P();
4     Q q1 = p.m2();
5     Q q2 = p.m2(); }
6   void m2() {
7     return new Q();
8   } /*m2*/
9 } /*class D*/
```

- Recall: *valcs* $\equiv_{precision}$ *kcs*
- *kcs*: m2 is analyzed twice.
- *valcs*: m2 is analyzed once.
- Both report that q1 and q2 are aliases after line 5.
- With heap-cloning:
  - *kcs*: q1 and q2 are not aliases.
  - *valcs*: q1 and q2 are aliases.

# Surprise #1 with heap cloning: *kcsH* vs *valcsH*

Expectations reduce the joy, surprise enhances joy! - Anonymous

```
1 class D {
2   void m1() {
3     P p = new P();
4     Q q1 = p.m2();
5     Q q2 = p.m2(); }
6   void m2() {
7     return new Q();
8   } /*m2*/
9 } /*class D*/
```

- Recall: $valcs \equiv_{precision} kcs$
- *kcs*: m2 is analyzed twice.
- *valcs*: m2 is analyzed once.
- Both report that q1 and q2 are aliases after line 5.
- With heap-cloning:
  - *kcs*: q1 and q2 are not aliases.
  - *valcs*: q1 and q2 are aliases.
- $valcsH \leq_{precision} kcsH$


# Surprise #2 with heap cloning: *lsrvH* vs *valcsH*

I love surprises as long as I like the outcome! - Anonymous

- Recall: $lsrv \equiv_{precision} valcs$
- $lsrvH \leq_{precision} valcsH^*$

*Reasoning in the paper.


# Surprise #2 with heap cloning: *lsrvH* vs *valcsH*

I love surprises as long as I like the outcome! - Anonymous

- Recall: $lsrv \equiv_{precision} valcs$
- $lsrvH \leq_{precision} valcsH^{*} \leq_{precision} kcsH$

*Reasoning in the paper.


# Surprise #2 with heap cloning: *lsrvH* vs *valcsH*

I love surprises as long as I like the outcome! - Anonymous

- Recall: $lsrv \equiv_{precision} valcs$
- $lsrvH \leq_{precision} valcsH^{*} \leq_{precision} kcsH$

**Moreover:**

- Recall: $lsrv \not\equiv_{precision} kobj$
- $lsrvH \not\equiv_{precision} kobjH^{*}$ (incomparable).

*Reasoning in the paper.


# Our idea: Mix the contexts

An insight has little value till it leads to a workable idea – Anonymous

**Insight:**

- Heap cloning alters the precision relations quite a bit.
- Existing context abstractions miss cases covered by each other
- Some approaches (e.g., LSRV variants) scale very well.
- **Q:** Why not use the abstractions together?

**Idea:**

- Merge abstractions $c_1$ and $c_2$ to get $c_{1\bullet 2}$ such that $c_{1\bullet 2}$ covers the optimization opportunities covered by both $c_1$ and $c_2$.
- In the resultant version, a method is analyzed if any of $c_1$ or $c_2$ is different from the previous contexts.


# Choosing the right mix

You can make a great fashion statement by doing the right mix and match – Anonymous

- More different the abstractions are, better may be the *precision* of the mix.
  $\Longrightarrow$ Choose one approach each from call-site- and object-sensitive approaches.
- The combined approach needs to be *scalable*.
  $\Longrightarrow$ Choose *lsrvH* from the call-site- group and *kobjH* from the object-sensitive group.

Obtained mix: **lsrv*k*objH**

# Choosing the right mix

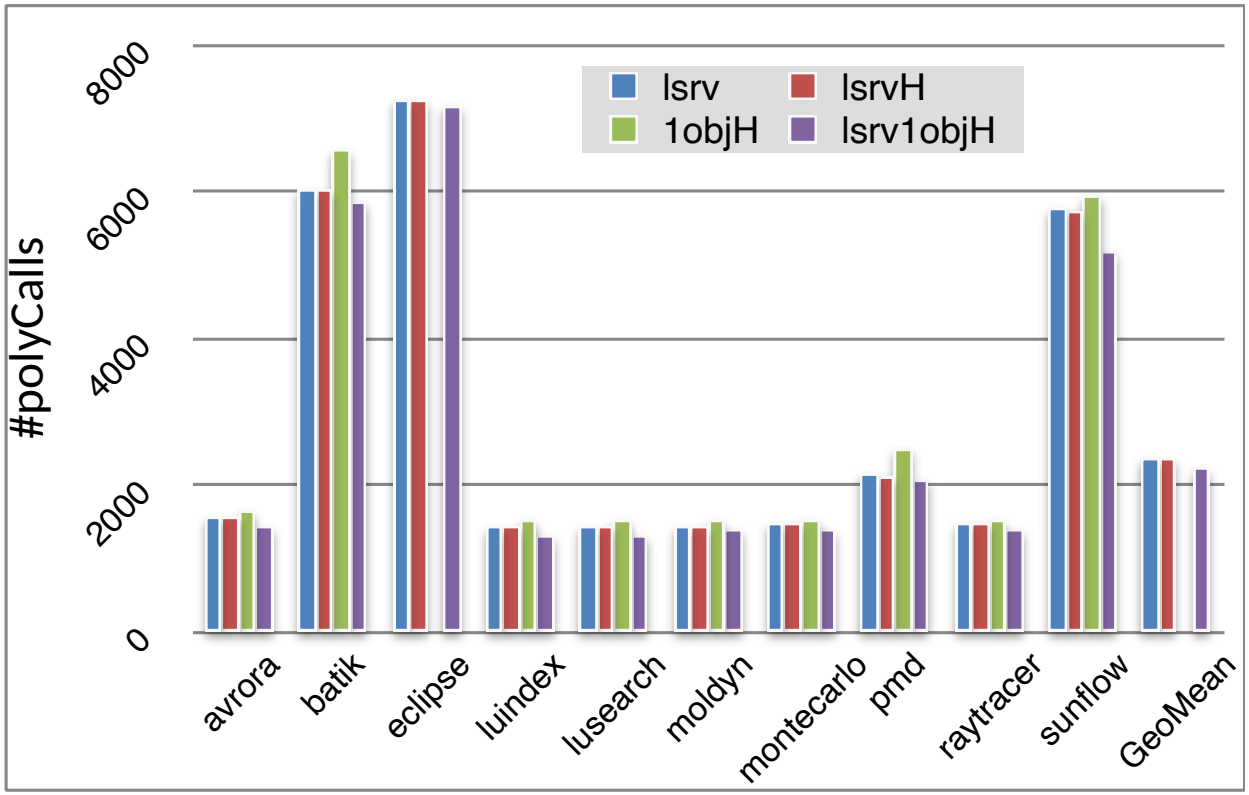
You can make a great fashion statement by doing the right mix and match – Anonymous

- More different the abstractions are, better may be the *precision* of the mix.
  $\Longrightarrow$ Choose one approach each from call-site- and object-sensitive approaches.
- The combined approach needs to be *scalable*.
  $\Longrightarrow$ Choose *lsrvH* from the call-site- group and *kobjH* from the object-sensitive group.

Obtained mix: **lsrv*k*objH**; Implemented in the Soot framework.

# Evaluation results (Devirtualization)

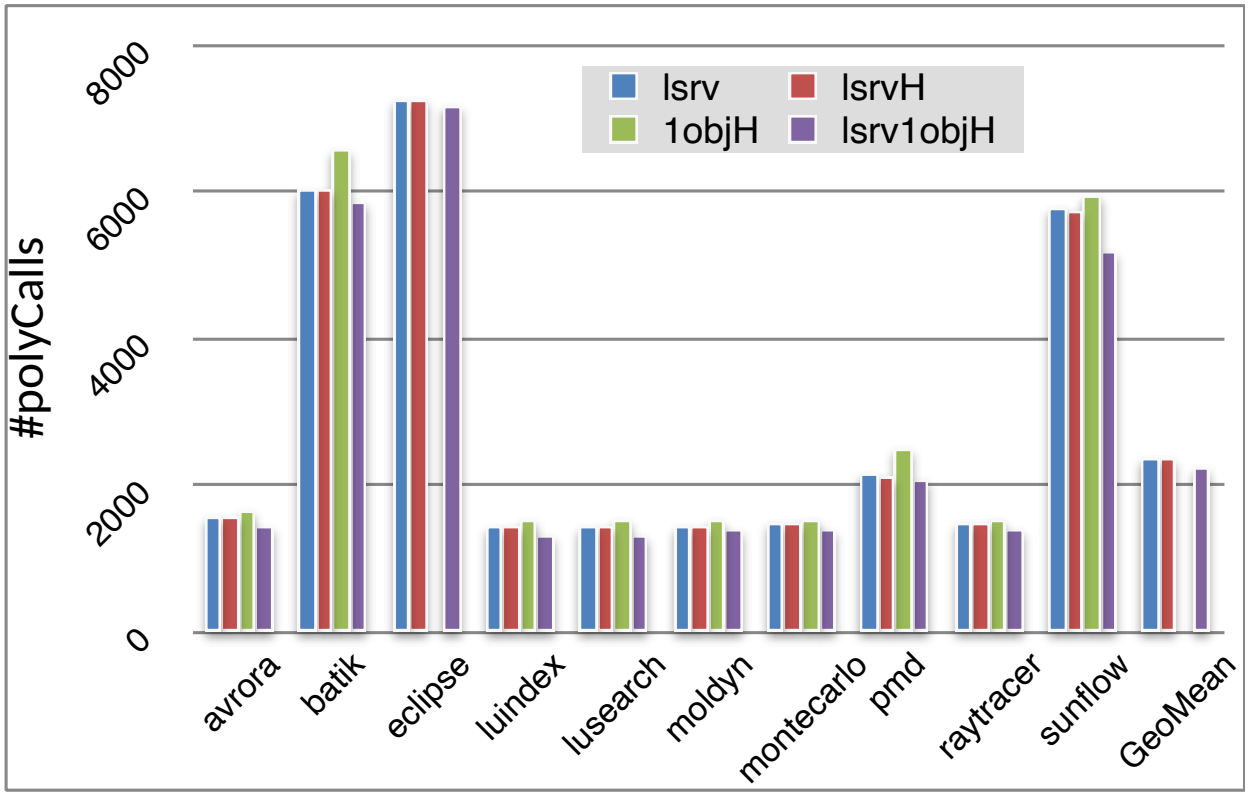
I evaluate, therefore I am. – System designer's mantra




lsrv*k*objH resolves the **least** number of calls as polymorphic.

# Evaluation results (Devirtualization)

I evaluate, therefore I am. – System designer's mantra




lsrv*k*objH resolves the **least** number of calls as polymorphic.

lsrv*k*objH leads to the **least** number of call-graph edges.

# Evaluation results (Time taken)

I evaluate, therefore I am. – System designer's mantra

| Benchmark | time | % increase |
|---|---|---|
| | *lsrv* (s) | *lsrvh* |
| avrora | 55 | 24.5 |
| batik | 946 | 167.3 |
| eclipse | 988 | 224.0 |
| luindex | 46 | 38.5 |
| lusearch | 57 | 44.6 |
| moldyn | 53 | 85.3 |
| montecarlo | 53 | 58.3 |
| pmd | 108 | 44.8 |
| raytracer | 53 | 62.1 |
| sunflow | 684 | 40.8 |
| GeoMean | 130 | 62.3 |


lsrv*k*objH scales well for all the benchmarks
(and improves precision as shown in the previous slides).

# Evaluation results (Time taken)

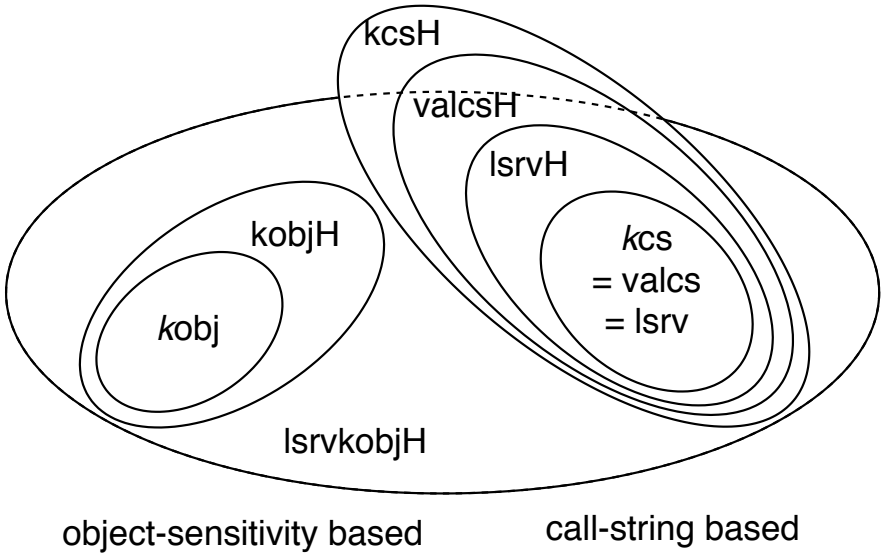
I evaluate, therefore I am. – System designer's mantra

| **Benchmark** | **time** | **% increase** | | |
|---|---|---|---|---|
| | *lsrv* (s) | *lsrvh* | 1*objh* | *lsrv*1*objh* |
| avrora | 55 | 24.5 | 646.2 | 26.5 |
| batik | 946 | 167.3 | 709.8 | 129.2 |
| eclipse | 988 | 224.0 | - | 225.9 |
| luindex | 46 | 38.5 | 653.9 | 23.3 |
| lusearch | 57 | 44.6 | 672.1 | 61.4 |
| moldyn | 53 | 85.3 | 448.5 | 83.6 |
| montecarlo | 53 | 58.3 | 474.3 | 67.2 |
| pmd | 108 | 44.8 | 587.3 | 2263.2 |
| raytracer | 53 | 62.1 | 452.6 | 68.7 |
| sunflow | 684 | 40.8 | 1097.1 | 53.2 |
| GeoMean | 130 | 62.3 | - | 93.6 |


lsrv*k*objH scales well for all the benchmarks
(and improves precision as shown in the previous slides).

# Updated world view

By the time you update your working space, it becomes outdated – Anonymous




- Newer variants find proper placement.
- Effects of heap cloning incorporated.
- lsrv*k*objH connects the previously unconnected approaches!

# In the paper

World is unlimited: talk $\rightarrow$ paper $\rightarrow$ matrix! – Anonymous

- Insights on reducing the overheads of object-sensitivity.
- 3-stage efficient computation of Isrv*k*objH contexts.
- Theoretical and practical precisions of existing and recent approaches.
- Correctness and termination discussions.
- Study of the memory consumption of the various approaches.

# Representative related work

Learn from those who have trodden the path before. – Explorer Anonymous

Explaining abstractions:

- Kanvar and Khedker (CS'16) survey heap abstractions and assert their importance towards precision and scalability.
- Smaragdakis et al. (POPL'11) clarify the definition of object-sensitivity and propose type-sensitivity as a close sibling.

Combining analyses/abstractions:

- Codish et al. (TOPLAS'95) perform multiple program analyses together over a combined domain.
- Kastrinis et al. (PLDI'13) propose hybrid context-sensitivity, but conclude that combining call-site- and object-sensitivity is impractical.


# Conclusion and Future Work

Conclusion = What we could do. Future work = What we could not. –Anonymous

- Clarified: relative precision of existing and recent context abstractions.
- Demonstrated: heap cloning leads to surprising precision relations.
- Introduced: novel idea of mixing abstractions to improve precision.
- Showed: mixing offers the best precision-scalability trade-off
  - gives benefits of both call-site- and object-sensitive approaches.

**Future Work:**

- Generalize mixing to multiple context abstractions and use it for different analyses.


# Questions?

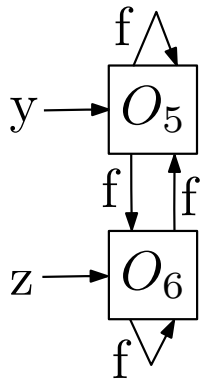

*Expected answers are not guaranteed.*

- Clarified: relative precision of existing and recent context abstractions.
- Demonstrated: heap cloning leads to surprising precision relations.
- Introduced: novel idea of mixing abstractions to improve precision.
- Showed: mixing offers the best precision-scalability trade-off
  - gives benefits over both call-site- and object-sensitive approaches.

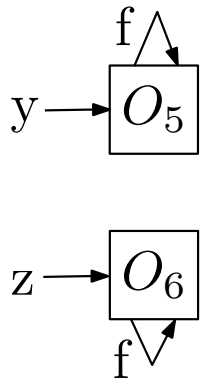
# Backup

# Incomparability of the two broad variants

```
1  class D {
2    ...
3    void foo() {
4      B x = new B();
5      Y y = new Y();
6      Z z = new Z();
7      x.m3(y);
8      x.m3(z);
9    }
10   B m3(B p) {
11     p.f = p;
12   }
13 }
```




*kobj*

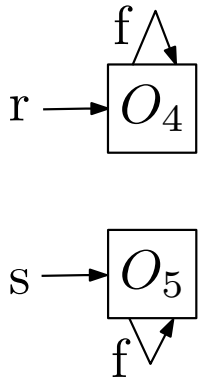



*kcs*/*valcs*/*lsrv*

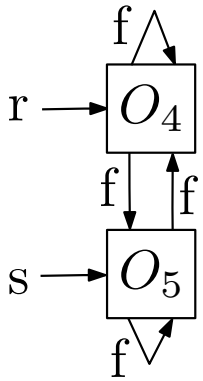
`y.f` and `z.f` are aliases in *kobj* (imprecise), but not in *kcs*/*valcs*/*lsrv* (precise).

# Incomparability of the two broad variants (Cont.)

```
1  class D {
2    ...
3    void bar() {
4      B r = new B();
5      B s = new B();
6      B t = * ? r : s;
7      t.m4();
8    }
9    B m4() {
10     this.f = this;
11   }
12 }
```




*kobj*




*kcs*/*valcs*/*lsrv*

`r.f` and `s.f` are aliases in *kcs*/*valcs*/*lsrv* (imprecise) but not in *kobj* (precise).

# Surprise #2 with heap cloning: *lsrvH* vs *valcsH*

```
 1 class W {
 2   X f;
 3   W() { f = new X(); }
 4   void setG(Y y) {
 5     f.g = y; }
 6   Y getG() {
 7     return f.g; } }
 8 class X { Y g; }
 9 class Y {
10   void m() {...} }
11 class Z extends Y {
12   void m() {...} }
```

```
13 class D {
14   void bar() {
15     W w1 = new W();
16     Y y1 = new Y();
17     w1.setG(y1);
18     W w2 = new W();
19     Z z1 = new Z();
20     w2.setG(z1);
21     Y p = w1.getG();
22     p.m();
23     Y q = w2.getG();
24     q.m(); } }
```

- *valcsH* re-analyzes `W`'s constructor and resolves the calls to `m` as monomorphic (unlike *lsrvH*).


# Surprise #3 with heap cloning: *lsrvH* vs *kobjH*

```
 1 class W {
 2   X f;
 3   W() { f = new X(); }
 4   void setG(Y y) {
 5     f.g = y; }
 6   Y getG() {
 7     return f.g; } }
 8 class X { Y g; }
 9 class Y {
10   void m() {...} }
11 class Z extends Y {
12   void m() {...} }
```

```
13  class D {
14    void bar() {
15      W w1 = new W();
16      Y y1 = new Y();
17      w1.setG(y1);
18      W w2 = new W();
19      Z z1 = new Z();
20      w2.setG(z1);
21      Y p = w1.getG();
22      p.m();
23      Y q = w2.getG();
24      q.m(); } }
```

- In *lsrvH*, `W`'s constructor is *not* re-analyzed at line 18, `w1.f` and `w2.f` point to $O_3$, `w1.f.g` and `w2.f.g` both point to $\{O_{16}, O_{19}\}$, and both the calls to `m` are polymorphic.

Not so in *kobjH*.