

Pick Your Contexts Well: Understanding Object-Sensitivity

The Making of a Precise and Scalable Pointer Analysis

Yannis Smaragdakis

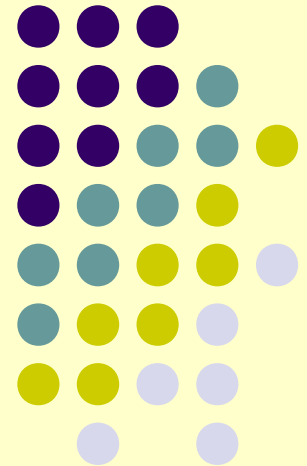
University of Massachusetts, Amherst
and University of Athens

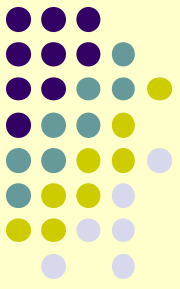
Martin Bravenboer

LogicBlox Inc.

Ondřej Lhoták

University of Waterloo

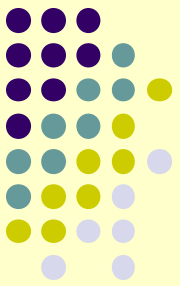




Context

- Object-sensitivity: an abstraction already behind the most precise and scalable points-to analyses
- Introduced by Milanova, Rountev and Ryder in 2002, quickly adopted in many practical settings
 - mostly for OO languages
- Still not completely understood:
 - the design space yields algorithms with very different precision
 - not clear how context affects precision and scalability

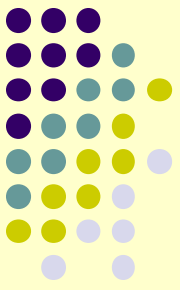




What is this paper about?

- We offer a clearer understanding of object-sensitivity design space, tradeoffs
- We exploit it to produce better points-to analysis: type-sensitive analysis
 - like object sensitive, but with some contexts replaced by types
 - choice matters a lot!
- Why do you care?
 - because there are some really cool insights
 - easy to follow
 - because the result is practical: currently the best tradeoff of precision and performance





First: what is points-to analysis?

- Static analysis: what objects can a variable point to?
- Highly recursive

```
class A {  
  void foo(Object o) {...}  
}
```

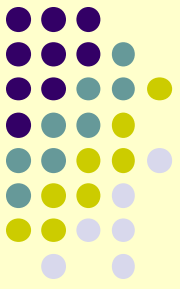
```
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

foo's o can point to whatever
someobj1 can point to

foo's o can point to whatever
someobj2 can point to



Call-site-sensitive points-to analysis / kCFA



- Typically made precise using “context”: e.g., call-sites

```
class A {  
  void foo(Object o) {...}  
}
```

```
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

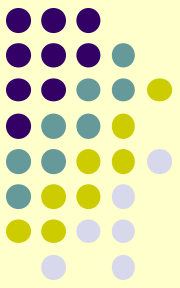
foo analyzed separately:

- once for o pointing to whatever someobj1 can point to
- once for o pointing to whatever someobj2 can point to

Important because of further analysis inside foo



In this talk: different context abstraction! Object-Sensitivity



- Object-sensitivity: information on objects used as context

```
class A {  
  void foo(Object o) {...}  
}  
  
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

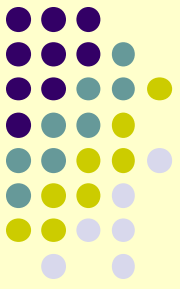
foo analyzed separately

- for each object pointed to by a1
 - for each object pointed to by a2
- How many cases in total?

0? 1? 2? ... 1million?

The number of contexts depends on the analysis so far!





A large design space

- What “information on objects used as context” ?

```
class A {  
  void foo(Object o) {...}  
}
```

```
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

Information available on an object:

- its creation site (instruction)
- the context for its creation site

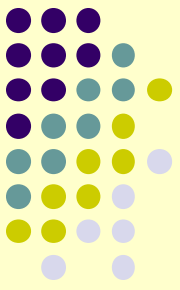
No matter what “context” is!

Context for a call has to be created out of:

- information for receiver object
- current context at call-site
- (information for caller object)

Need to at least collapse two contexts into one

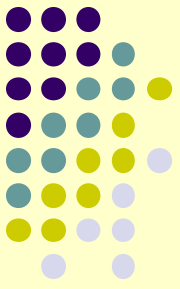




Design Space

- This choice (practically $\binom{2n}{n}$ options) has not been acknowledged before
- The choices made by standard published algorithms and implementations vary widely
 - mostly without realizing
- The result is completely different precision and performance





Example: Paddle vs. Milanova

- For a 2-object-sensitive analysis: context is 2 allocation sites

```
class A {  
  void foo(Object o) {...}  
}  
  
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

Original object-sensitivity
(Milanova) uses:

- receiver (a1 or a2) allocation site
- allocation site of receiver's allocator

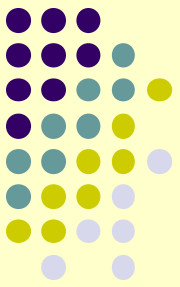
PADDLE framework uses:

- receiver (a1 or a2) allocation site
- caller allocation site
 - *i.e., of a **Client** object, not an **A** object*

Quiz: which one do we think wins?



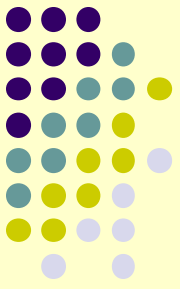
General formal framework for context-sensitive analyses



- Keep context-sensitive variables, a store, sets $Context$, $HContext$,
 - abstr. interpretation over A-Normal FJ formalism [Might, Smaragdakis, and Van Horn@PLDI'10]
- Functions:
 - $record: Instr \times Context \rightarrow HContext$
 - $merge: Instr \times HContext \times Context \rightarrow Context$
- Key analysis logic:
 - $i: [v = new C();]$ with context $c \rightarrow$ store heap context $record(i,c)$ with v
 - $i: [v.m(...);]$ with context $c \rightarrow$ analyze m with context $merge(i,hc,c)$ where hc is the context stored with v



We can now express past analyses nicely

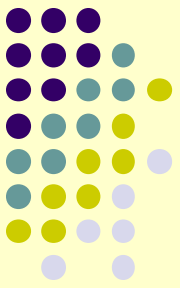


- Original Milanova et al.-style object-sensitivity:
 - $Context = HContext = Instr^n$
- Functions:
 - $record(i, c) = cons(i, first_{n-1}(c))$
 - $merge(i, hc, c) = hc$

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new\ C();]$ with context $c \rightarrow$
store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$
analyze m with context $merge(i, hc, c)$ where hc
is the context stored with v



We can now express past analyses nicely

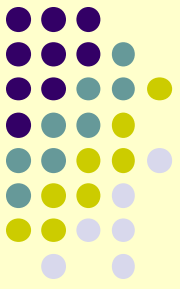


- Paddle-style object-sensitivity:
 - $Context = HContext = Instr^n$
- Functions:
 - $record(i, c) = cons(i, first_{n-1}(c))$
 - $merge(i, hc, c) = cons(car(hc), first_{n-1}(c))$

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new\ C();]$ with context $c \rightarrow$
store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$
analyze m with context $merge(i, hc, c)$ where hc
is the context stored with v



We can now express past analyses nicely

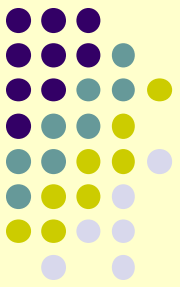


- Most commonly called “object-sensitivity”:
 - $HContext = Instr, Context = Instr^n$
- Functions:
 - $record(i, c) = i$
 - $merge(i, hc, c) = cons(hc, first_{n-1}(c))$

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new C();]$ with context $c \rightarrow$ store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$ analyze m with context $merge(i, hc, c)$ where hc is the context stored with v



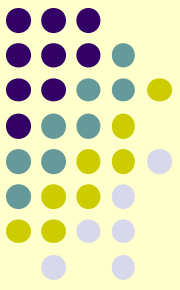
We can now express past analyses nicely



- object-sensitive+H analyses (*heap cloning*):
 - $HContext = Instr^{n+1}$, $Context = Instr^n$
- Functions:
 - $record(i, c) = cons(i, c)$
 - $merge(i, hc, c) =$ [any of the previous options]

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new\ C();]$ with context $c \rightarrow$ store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$ analyze m with context $merge(i, hc, c)$ where hc is the context stored with v



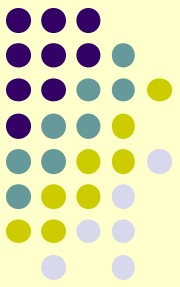


Some insights on context

- When context consists of n elements with K possibilities for each, we analyze each method up to nK times
 - e.g., $K = \#allocation\ sites$
- Relative to a shallower context (e.g., $n-1$) we may replicate same points-to data K times
- Ideal for precision: extra context elements partition space into small sets, i.e., evenly
- I.e., context elements are uncorrelated
 - otherwise combinations uneven



Revisit Example: Paddle vs. Milanova



- For a 2-object-sensitive analysis: context is 2 allocation sites

```
class A {  
  void foo(Object o) {...}  
}
```

```
class Client {  
  void bar(A a1, A a2) {  
    ...  
    a1.foo(someobj1);  
    ...  
    a2.foo(someobj2);  
  }  
}
```

Original obj.-sens. (Milanova) uses:

- receiver (a1 or a2) allocation site
- allocation site of receiver's allocator

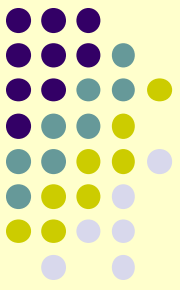
PADDLE framework uses:

- receiver (a1 or a2) allocation site
- caller allocation site

Quiz: which one do we think wins?

- ***Original. Receiver and caller are highly correlated!***
- e.g., same object, wrapper object, design patterns

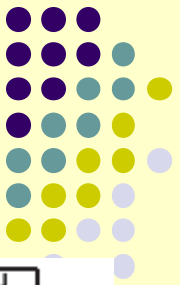




A significant difference

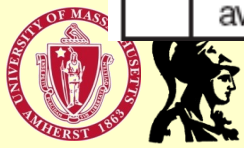
- Good choice of context is more precise:
 - smaller points-to sets
 - better results for client analyses: static cast elimination, de-virtualization, reachable methods
 - often difference on 2-object-sensitive analyses (good vs. bad context) as great as from 1-object-sensitive
- Good choice of context yields much faster implementation!
 - often 2x or more
 - using our **DOOP** framework



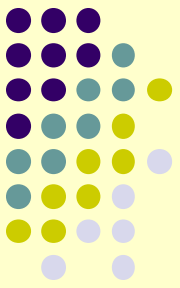


A significant difference

		insensitive	1obj	1obj+H	2plain+1H	2full+1H
andr	call-graph edges	43055	-559	-1216	-1129	-368
	reachable methods	5758	-29	-37	-62	-21
	total reachable virtual call sites	27823	-128	-96	-272	-139
	total polymorphic call sites	1326	-38	-22	-38	-68
	application reachable virtual call sites	16393	0	0	0	-9
	application polymorphic call sites	851	0	0	0	0
	total reachable casts	1038	-14	-15	-33	-6
	total casts that may fail	844	-136	-94	-144	-64
	application reachable casts	308	0	0	0	-1
	application casts that may fail	262	-8	-38	-66	-23
average var-points-to	216.71	24.7	15.1	8.5	8.2	
average application var-points-to	327.27	20.8	15.3	8.8	8.5	
chart	call-graph edges	44930	-1239	-2063	-2287	-765
	reachable methods	8502	-76	-87	-115	-53
	total reachable virtual call sites	23944	-233	-327	-368	-172
	total polymorphic call sites	1218	-90	-24	-83	-119
	application reachable virtual call sites	3649	0	-8	-47	-12
	application polymorphic call sites	110	-4	-13	-10	-4
	total reachable casts	1728	-22	-38	-58	-7
	total casts that may fail	1457	-182	-252	-164	-120
	application reachable casts	232	0	-4	-21	-1
	application casts that may fail	196	-17	-64	-32	-38
average var-points-to	98.35	36.0	20.1	9.4	6.7	
average application var-points-to	55.35	27.2	14.4	5.0	2.8	

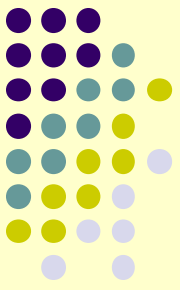


Some more understanding of contexts



- The problem with precise, deep-context analyses is that they *may* explode in complexity
 - when deeper context yields precision, it is great
 - even better performance
 - when imprecision creeps in, **scalability wall**: extra level of context, $O(K)$ multiplicative factor in complexity
 - plain combinatorial explosion
- Result: some programs are fast(er), some completely hopeless



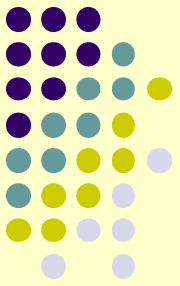


Idea: type-sensitivity

- Why not alleviate the combinatorial explosion by reducing combinations
- Instead of allocation sites, keep types
- Otherwise precisely isomorphic to object-sensitivity
 - just some elements of context are transformed by a function $\mathbf{T}: \text{Instr} \rightarrow \text{ClassName}$



Example type-sensitive analyses

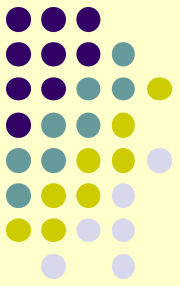


- 2type+1H:
 - $HContext = Instr \times ClassName$
 $Context = ClassName^2$
- Functions:
 - $record(i, [C_1, C_2]) = [i, C_1]$
 - $merge(i, [i', C], c) = [\mathbf{T}(i'), C]$

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new C();]$ with context $c \rightarrow$
store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$
analyze m with context $merge(i, hc, c)$ where hc
is the context stored with v



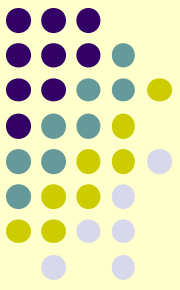
Example type-sensitive analyses



- 1type1obj+1H:
 - $HContext = Instr^2$
 $Context = Instr \times ClassName$
- Functions:
 - $record(i, [i', C]) = [i, i']$
 - $merge(i, [i_1, i_2], c) = [i_1, \mathbf{T}(i_2)]$

- $record: Instr \times Context \rightarrow HContext$
- $merge: Instr \times HContext \times Context \rightarrow Context$
- $i: [v = new C();]$ with context $c \rightarrow$
store heap context $record(i, c)$ with v
- $i: [v.m(...);]$ with context $c \rightarrow$
analyze m with context $merge(i, hc, c)$ where hc
is the context stored with v





What function **T** to choose?

```
class A {  
    ...  
    i: B b = new B();  
    ...  
    b.foo(...);  
}
```

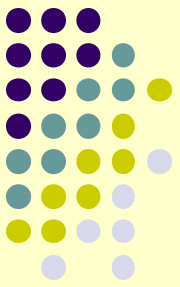
Which type gives more information about *i*? A or B?

i used in representing receiver object when analyzing specific implementation of method foo

B offers little info: we already know good upper bound for B when analyzing foo:

- either `B::foo` or `C::foo` for some close superclass C

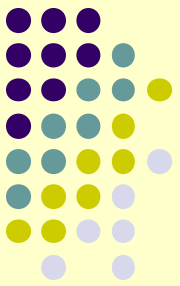




Type-sensitivity in practice

- Type-sensitive analyses work great in practice!
- Very fast, very few scalability issues
 - $2\text{type}+1\text{H}$ at least 2x (and up to 8x) faster than $1\text{obj}+\text{H}$ for 9 out of 10 DaCapo benchmarks
 - while almost always much more precise
 - an excellent approximation of full object-sensitive analyses
- $2\text{type}+1\text{H}$ is probably the new sweet spot for a practical precise analysis





Conclusions

- We offered a clearer understanding of object-sensitivity design space, tradeoffs
- We exploited it to produce better points-to analysis: type-sensitive analysis
 - like object sensitive, but with some contexts replaced by types
 - choice matters a lot!
- Why do you care?
 - because there are some really cool insights
 - easy to follow
 - because the result is practical: currently the best tradeoff of precision and performance

