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

```java
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

```java
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

```java
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? ... 1 million?

*The number of contexts depends on the analysis so far!*
A large design space

- What “information on objects used as context”?

```java
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

```java
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 x Context \rightarrow HContext
  - merge: Instr x HContext x Context \rightarrow Context
- Key analysis logic:
  - \( i: [v = \text{new} \ C(); ] \) with context \( c \) \rightarrow store heap context \( \text{record}(i,c) \) with \( v \)
  - \( i: [v.m(...); ] \) with context \( c \) \rightarrow analyze \( m \) with context \( \text{merge}(i,hc,c) \) where \( hc \) is the context stored with \( v \)
We can now express past analyses nicely

- Original Milanova et al.-style object-sensitivity:
  - \( \text{Context} = H\text{Context} = \text{Instr}^n \)
- Functions:
  - \( \text{record}(i, c) = \text{cons}(i, \text{first}_{n-1}(c)) \)
  - \( \text{merge}(i, hc, c) = hc \)

- \( \text{record}: \text{Instr} \times \text{Context} \rightarrow H\text{Context} \)
- \( \text{merge}: \text{Instr} \times H\text{Context} \times \text{Context} \rightarrow \text{Context} \)
- \( i: [\nu = \text{new} \ C(); \ ] \) with context \( c \) ➔
  store heap context \( \text{record}(i,c) \) with \( \nu \)
- \( i: [\nu.m(...); \ ] \) with context \( c \) ➔
  analyze \( m \) with context \( \text{merge}(i, hc, c) \) where \( hc \) is the context stored with \( \nu \)
We can now express past analyses nicely

- **PADDLE-style object-sensitivity:**
  - \( \text{Context} = H\text{Context} = \text{Instr}^n \)

- Functions:
  - \( \text{record}(i,c) = \text{cons}(i, \text{first}_{n-1}(c)) \)
  - \( \text{merge}(i, hc, c) = \text{cons}(\text{car}(hc), \text{first}_{n-1}(c)) \)

- **record**: \( \text{Instr} \times \text{Context} \rightarrow H\text{Context} \)
- **merge**: \( \text{Instr} \times H\text{Context} \times \text{Context} \rightarrow \text{Context} \)
- \( i: [v = \text{new C}(); \ ] \text{with context } c \) ➔ store heap context \( \text{record}(i,c) \) with \( v \)
- \( i: [v.m(...); ] \text{ with context } c \) ➔ analyze \( m \) with context \( \text{merge}(i,hc,c) \) 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:
  - \(record(i,c) = \text{cons}(i, c)\)
  - \(merge(i, hc, c) = \text{[any of the previous options]}\)

  - \(record: \text{Instr} \times \text{Context} \rightarrow HContext\)
  - \(merge: \text{Instr} \times HContext \times \text{Context} \rightarrow \text{Context}\)
  - \(i: [\nu = \text{new } C(); \text{ ] with context } c \Rightarrow \text{store heap context } record(i, c) \text{ with } \nu\)
  - \(i: [\nu.m(...); \text{ ] with context } c \Rightarrow \text{analyze } m \text{ with context } merge(i, hc, c) \text{ where } hc \text{ is the context stored with } \nu\)
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., \( K = \#\text{allocation sites} \)
- 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

```java
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

<table>
<thead>
<tr>
<th></th>
<th>insensitive</th>
<th>1obj</th>
<th>1obj+H</th>
<th>2plain+1H</th>
<th>2full+1H</th>
</tr>
</thead>
<tbody>
<tr>
<td>call-graph edges</td>
<td>43055</td>
<td>-559</td>
<td>-1216</td>
<td>-1129</td>
<td>-368</td>
</tr>
<tr>
<td>reachable methods</td>
<td>5758</td>
<td>-29</td>
<td>-37</td>
<td>-62</td>
<td>-21</td>
</tr>
<tr>
<td>total reachable virtual call sites</td>
<td>27823</td>
<td>-128</td>
<td>-96</td>
<td>-272</td>
<td>-139</td>
</tr>
<tr>
<td><strong>total polymorphic call sites</strong></td>
<td><strong>1326</strong></td>
<td><strong>-38</strong></td>
<td><strong>-22</strong></td>
<td><strong>-38</strong></td>
<td><strong>-68</strong></td>
</tr>
<tr>
<td>application reachable virtual call sites</td>
<td>16393</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-9</td>
</tr>
<tr>
<td>application polymorphic call sites</td>
<td>851</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>total reachable casts</td>
<td>1038</td>
<td>-14</td>
<td>-15</td>
<td>-33</td>
<td>-6</td>
</tr>
<tr>
<td><strong>total casts that may fail</strong></td>
<td><strong>844</strong></td>
<td><strong>-136</strong></td>
<td><strong>-94</strong></td>
<td><strong>-144</strong></td>
<td><strong>-64</strong></td>
</tr>
<tr>
<td>application reachable casts</td>
<td>308</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>application casts that may fail</td>
<td>262</td>
<td>-8</td>
<td>-38</td>
<td>-66</td>
<td>-23</td>
</tr>
</tbody>
</table>

**Average var-points-to**

| antr                      | 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      |

<table>
<thead>
<tr>
<th></th>
<th>insensitive</th>
<th>1obj</th>
<th>1obj+H</th>
<th>2plain+1H</th>
<th>2full+1H</th>
</tr>
</thead>
<tbody>
<tr>
<td>call-graph edges</td>
<td>44930</td>
<td>-1239</td>
<td>-2063</td>
<td>-2287</td>
<td>-765</td>
</tr>
<tr>
<td>reachable methods</td>
<td>8502</td>
<td>-76</td>
<td>-87</td>
<td>-115</td>
<td>-53</td>
</tr>
<tr>
<td>total reachable virtual call sites</td>
<td>23944</td>
<td>-233</td>
<td>-327</td>
<td>-368</td>
<td>-172</td>
</tr>
<tr>
<td><strong>total polymorphic call sites</strong></td>
<td><strong>1218</strong></td>
<td><strong>-90</strong></td>
<td><strong>-24</strong></td>
<td><strong>-83</strong></td>
<td><strong>-119</strong></td>
</tr>
<tr>
<td>application reachable virtual call sites</td>
<td>3649</td>
<td>0</td>
<td>-8</td>
<td>-47</td>
<td>-12</td>
</tr>
<tr>
<td>application polymorphic call sites</td>
<td>110</td>
<td>-4</td>
<td>-13</td>
<td>-10</td>
<td>-4</td>
</tr>
<tr>
<td>total reachable casts</td>
<td>1728</td>
<td>-22</td>
<td>-38</td>
<td>-58</td>
<td>-7</td>
</tr>
<tr>
<td><strong>total casts that may fail</strong></td>
<td><strong>1457</strong></td>
<td><strong>-182</strong></td>
<td><strong>-252</strong></td>
<td><strong>-164</strong></td>
<td><strong>-120</strong></td>
</tr>
<tr>
<td>application reachable casts</td>
<td>232</td>
<td>0</td>
<td>-4</td>
<td>-21</td>
<td>-1</td>
</tr>
<tr>
<td>application casts that may fail</td>
<td>196</td>
<td>-17</td>
<td>-64</td>
<td>-32</td>
<td>-38</td>
</tr>
</tbody>
</table>

**Average var-points-to**

| chart                     | 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 isomorphically to object-sensitivity
  - just some elements of context are transformed by a function $\mathcal{T}: \text{Instr} \rightarrow \text{ClassName}$
Example type-sensitive analyses

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

- 1type1obj+1H:
  - $HContext = \text{Instr}^2$
  - $Context = \text{Instr} \times \text{ClassName}$

- Functions:
  - $record(i, [i',C]) = [i,i']$
  - $merge(i, [i_1,i_2], c) = [i_1, \mathcal{T}(i_2)]$

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

B offers little info: we already know good upper bound for B when analyzing \( \text{foo} \):
• either \( B::\text{foo} \) or \( C::\text{foo} \) for some close superclass C
Type-sensitivity in practice

- Type-sensitive analyses work great in practice!
- Very fast, very few scalability issues
  - 2type+1H at least 2x (and up to 8x) faster than 1obj+H for 9 out of 10 DaCapo benchmarks
  - while almost always much more precise
  - an excellent approximation of full object-sensitive analyses
- 2type+1H 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