OwnKit: Inferring Modularly Checkable Ownership Annotations for Java

Constantine Dymnikov, David J. Pearce and Alex Potanin

School of Engineering and Computer Science
Victoria University of Wellington

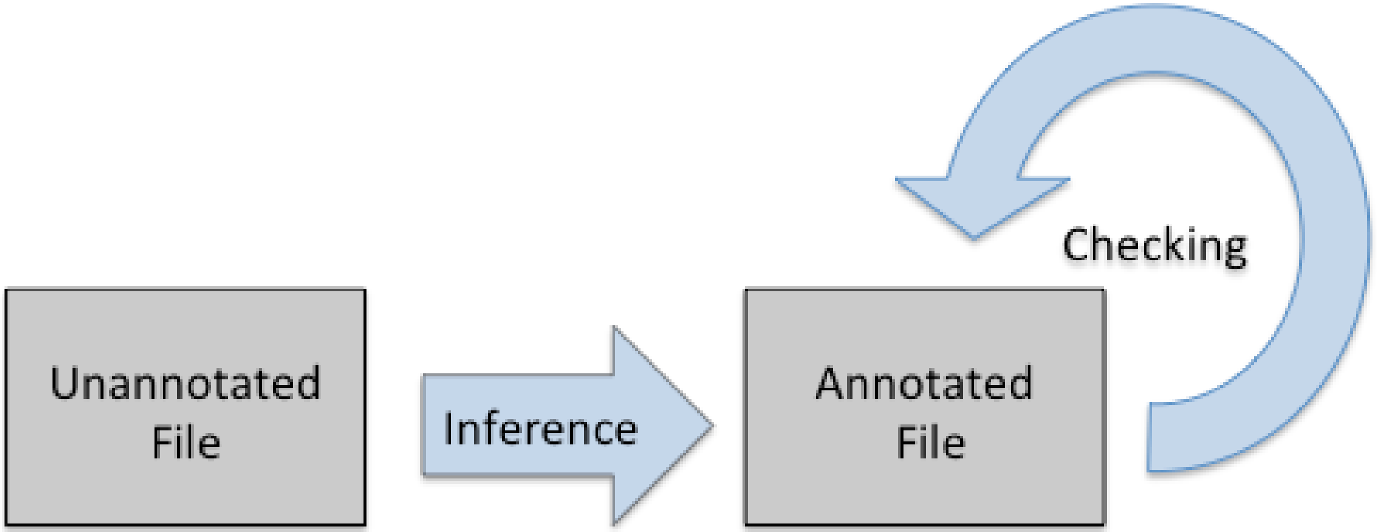
What is Ownership?

- Ownership is an approach for managing *aliasing*
- For example:

```
public class Rectangle {
    private @Owned List<Point> points;
    ...
}
```

- Objects reachable from `points` are **owned** (a.k.a *deep ownership*)
- Useful for lots of things! e.g. *parallelisation*, *verification*, and more.
- And, of course, it helps you find bugs in your code!

Ownership Inference versus Ownership Checking



- For this to work, inference can be **expensive**, but checking must be **efficient**.
- Ensuring annotations are **modularly checkable** is our approach.

What are Modularly Checkable Annotations?

Definition (Modular Type Checking)

A type checker is **modular** if checking a given module only requires access to the **type signatures** of external methods.

- This characterises **type checking** in Java (and other languages), as well as e.g. checking `final` class modifiers.
- This does not characterise **interprocedural** analysis, which requires access to all method bodies simultaneously.
- When checking a given method, signatures for other methods are **assumed correct**.

Overview

Definition (Object Graph)

An *object graph*, $\mathtt{O_G}$, is a directed graph capturing a snapshot of the heap at a given moment. Here, $\mathtt{o_1} \overset{\mathtt{C.f}}{\rightarrow} \mathtt{o_2} \in \mathtt{O_G}$ denotes that object $\mathtt{o_1}$ refers to object $\mathtt{o_2}$ via the field `f` declared in class `C`.

Definition (Ownership Guarantee)

Let `C.f` be a non-primitive field annotated with `@Owned` which is declared in class `C`. Then, for all objects $\mathtt{o_1}, \mathtt{o_2}, \mathtt{o_3}$ where $\mathtt{o_1} \overset{\mathtt{C.f}}{\rightarrow} \mathtt{o_3} \in \mathtt{O_G}$ and $\mathtt{o_2} \overset{\mathtt{C.f}}{\rightarrow} \mathtt{o_3} \in \mathtt{O_G}$ it follows that $\mathtt{o_1} = \mathtt{o_2}$.

- **Aim:** to *infer* which fields may safely be annotated `@Owned`
- **Approach:** determine which fields are *exposed*.
- **Assumptions:** parameters and return values for **`public`** or **`protected`** methods are *exposed*; fields declared **`public`** or **`protected`** are *exposed*.

Read Exposure

```
public class MyClass {
  private List<String> myList = ...;

  public List<String> getMyList(){
    return myList;
} }

public class External {
  public void expose(){
    MyClass mc = ...;
    List<String> alias = mc.getMyList();
    alias.add("bad");
} }
```

- A variable is **read exposed** if its value may be read externally.

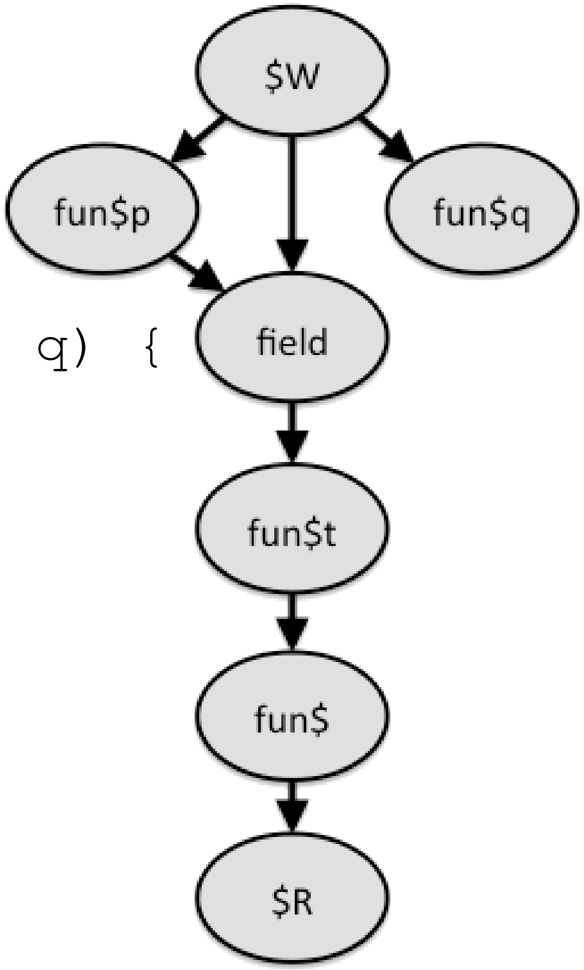
Write Exposure

```
public class MyClass {
  private List<String> myList = ...;

  public void setMyList(List<String> par){
    myList = par;
} }

public class External {
  public void expose() {
    MyClass mc = ...;
    List<String> alias = ...;
    mc.setMyList(alias);
} }
```

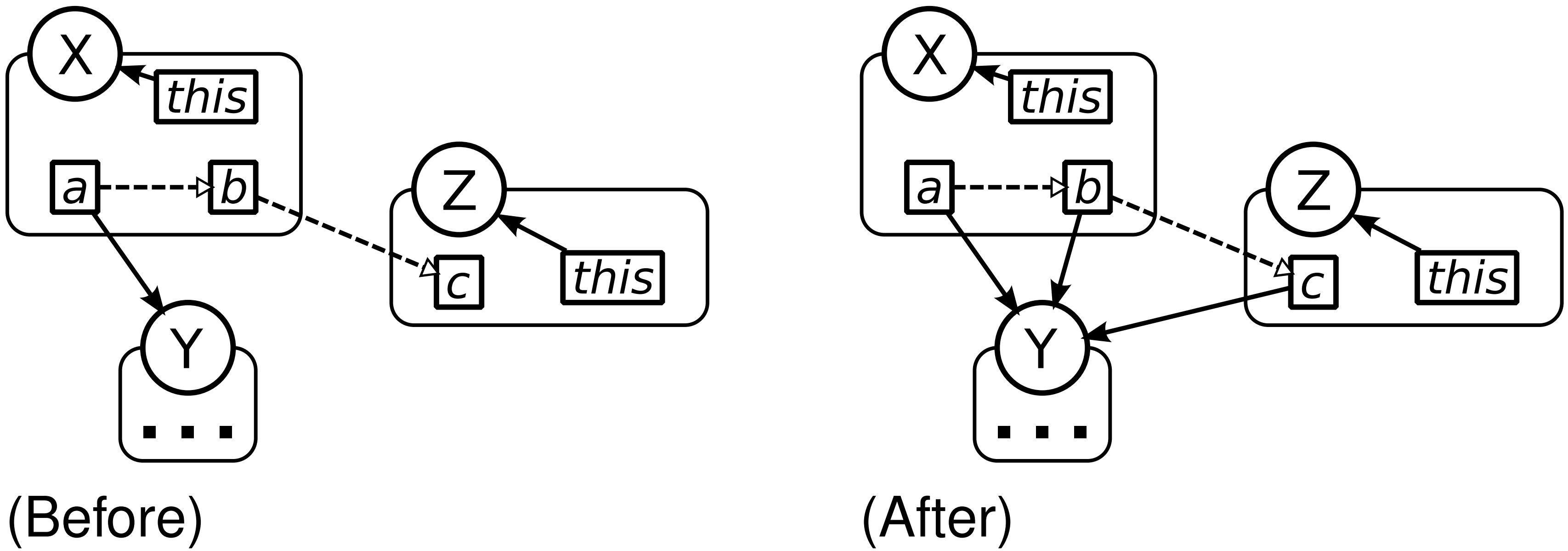
- A variable is **write exposed** if can be assigned a value that may be read externally.

Implementation — *Class Graph Construction*

```
public class MyClass {

 private Object field;

 public Object fun(Object p, MyClass q) {
   Object t = field;

   if(p!=null) {
     this.field = p;
    } else {
     this.field = q.field;
   }
   return t;
}}
```

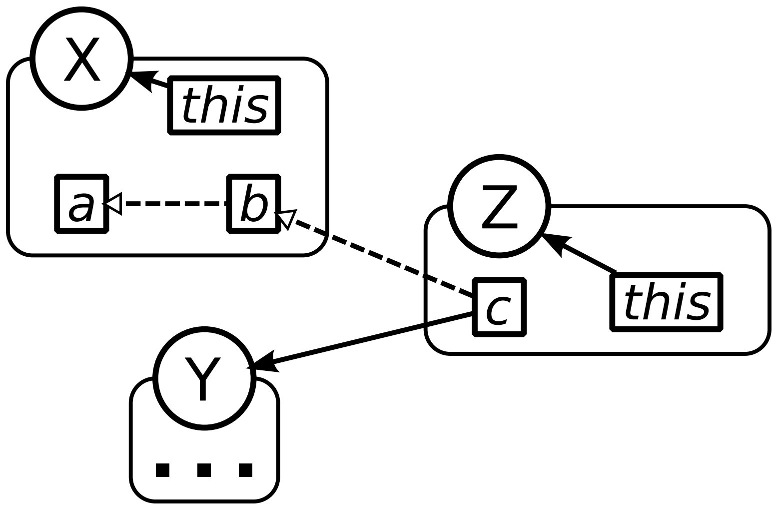


Implementation — *Exposure Propagation*

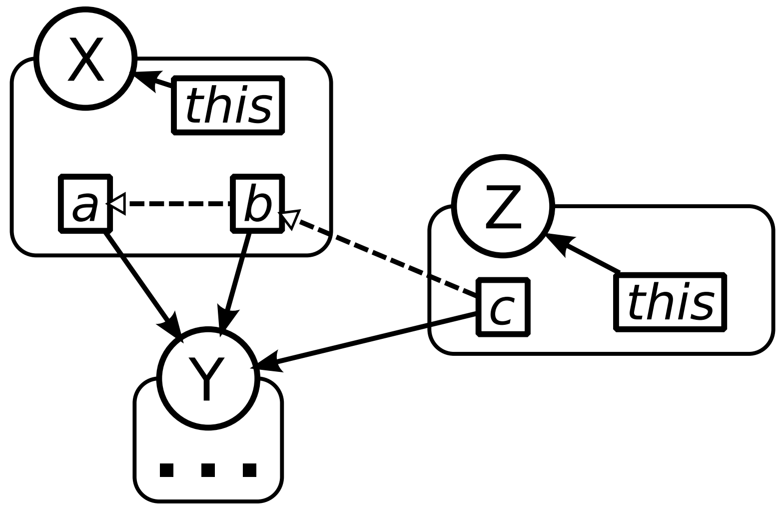


(case #1 — *read exposure*)

Implementation — *Exposure Propagation*



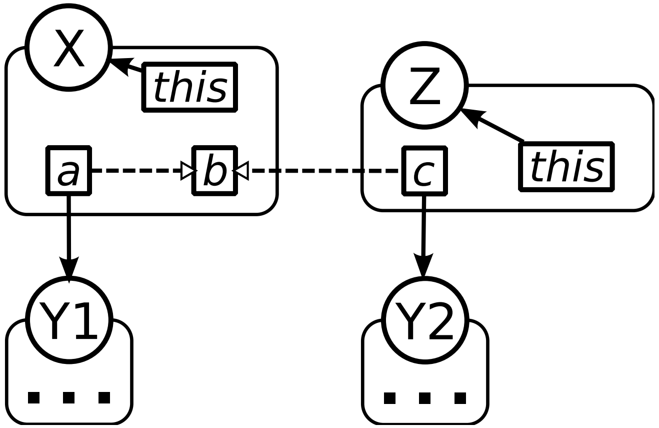
(Before)



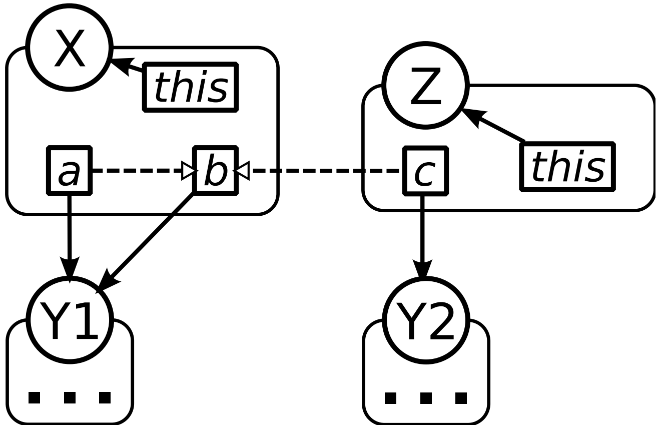
(After)

(case #2 — *write exposure*)

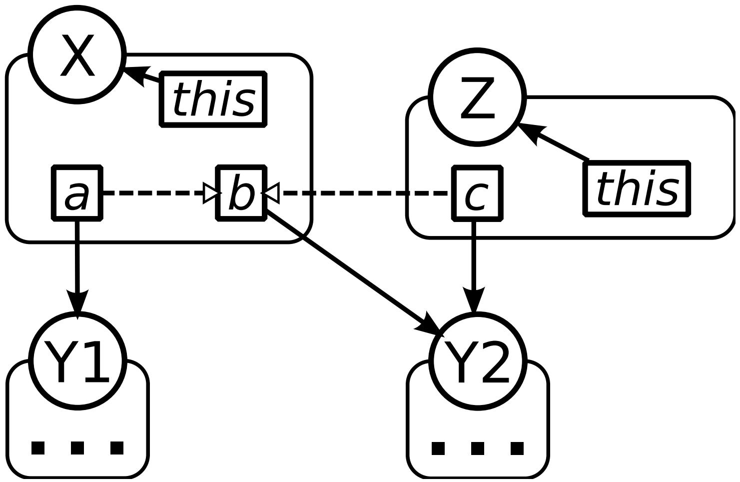
Implementation — *Exposure Propagation*



(Before)



(State 1)



(State 2)

Implementation — *Self Exposure*

- One complication for our analysis is **self exposure**:

```
public class Z {
  public Z(){
    S.staticField = this;
} }

public class S {
  public static Z staticField = ...;
}
```

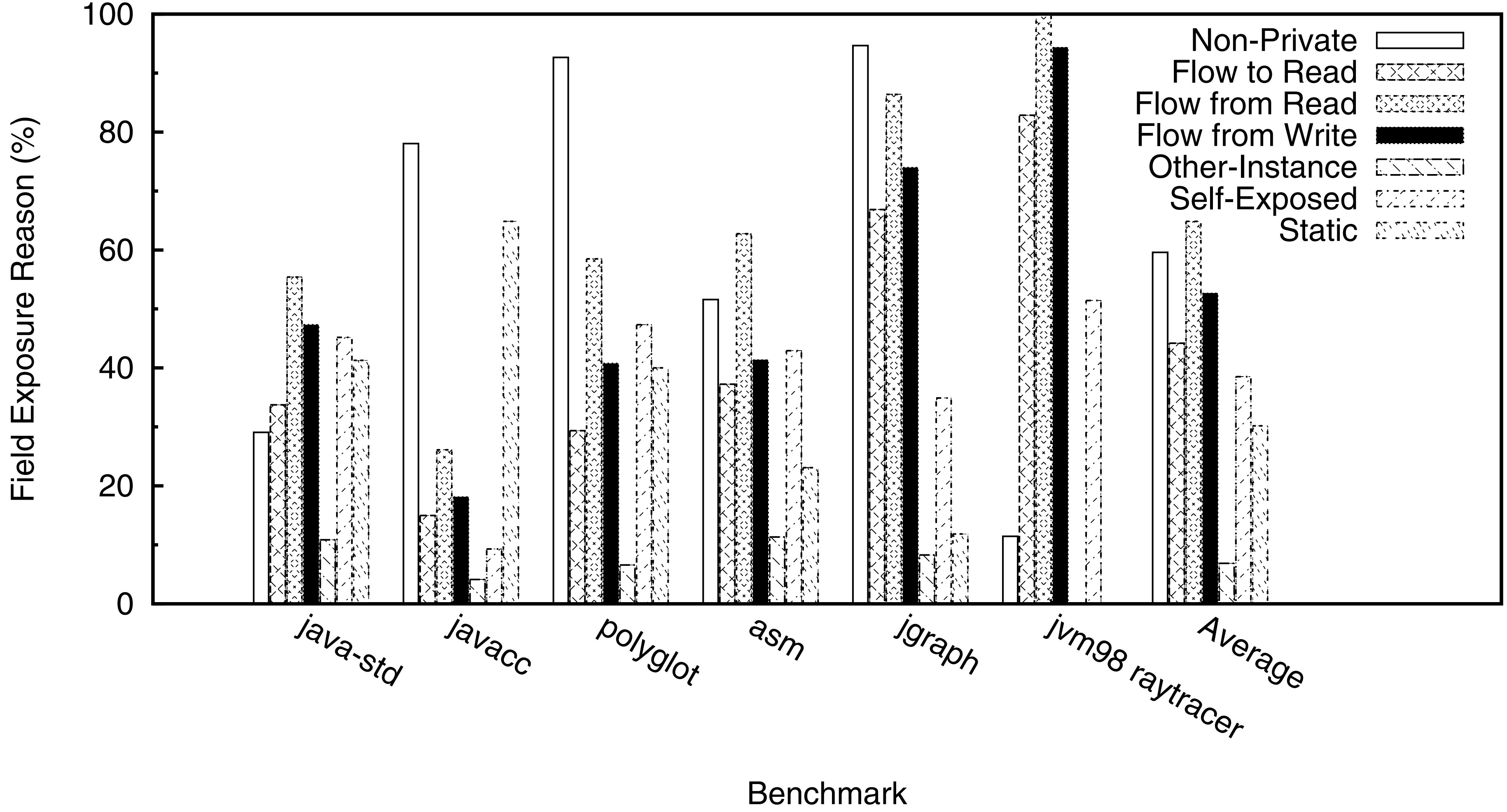
- Any variable that can reference objects of type `Z` are **read exposed**.

- In presence of self exposure, ownership remains modularly checkable with an **explicit annotation** (e.g. `@SelfExposed`).

Experimental Results — *How much Ownership?*

Program	**LOC**	**Total Fields**	**% of Owned Fields**		**Classes**	
			OwnKit	**UNO**	**Self-Exp. %**	**Total**
java-std	62,508	690	3.77	-	16.0	763
javacc	36,672	406	4.7	11.8	13.3	150
polyglot	14,148	421	0.5	2.9	11.0	327
asm	22,474	259	4.2	10.8	14.0	172
jgraph	12,262	178	5.1	3.9	29.2	89
raytracer	1,928	40	12.5	5.0	28.0	25
Average			**5.1**	**6.9**	**18.6**	

Experimental Results — *Reasons for Exposure*



Conclusion

- For ownership to be used, it needs to fit within **day-day development**.
- This means it must not impose **significant overhead**.
- Our approach is to focus on **modularly checkable** annotations:
  - Annotations are inferred using **expensive inference**.
  - Annotations are maintained using **efficient checker**.
- We presented a simple **ownership inference** scheme.
- Results are encouraging, compared with a heavy weight **interprocedural analysis**.