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:
  ```java
  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, $O_G$, is a directed graph capturing a snapshot of the heap at a given moment. Here, $o_1 \xrightarrow{C.f} o_2 \in O_G$ denotes that object $o_1$ refers to object $o_2$ 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 $o_1, o_2, o_3$ where $o_1 \xrightarrow{C.f} o_3 \in O_G$ and $o_2 \xrightarrow{C.f} o_3 \in O_G$ it follows that $o_1 = o_2$.

- **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.
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.
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 it can be assigned a value that may be read externally.
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*

(Before)

(After)

(case #1 — *read exposure*)
Implementation — *Exposure Propagation*

(Before)

(After)

(case #2 — write exposure)
Implementation — Exposure Propagation

(Before)

(State 1)

(State 2)
One complication for our analysis is self exposure:

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

```java
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?*

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Total Fields</th>
<th>% of Owned Fields</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>OwnKit</td>
<td>UNO</td>
</tr>
<tr>
<td>java-std</td>
<td>62,508</td>
<td>690</td>
<td>3.77</td>
<td>-</td>
</tr>
<tr>
<td>javacc</td>
<td>36,672</td>
<td>406</td>
<td>4.7</td>
<td>11.8</td>
</tr>
<tr>
<td>polyglot</td>
<td>14,148</td>
<td>421</td>
<td>0.5</td>
<td>2.9</td>
</tr>
<tr>
<td>asm</td>
<td>22,474</td>
<td>259</td>
<td>4.2</td>
<td>10.8</td>
</tr>
<tr>
<td>jgraph</td>
<td>12,262</td>
<td>178</td>
<td>5.1</td>
<td>3.9</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,928</td>
<td>40</td>
<td>12.5</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td>5.1</td>
<td>6.9</td>
</tr>
</tbody>
</table>
Experimental Results — *Reasons for Exposure*

![Graph showing field exposure reasons for different benchmarks. The graph compares non-private flows to read, flows from read, flows from write, other-instance exposure, self-exposed, and static exposure across various benchmarks such as Java-std, javacc, polyglot, asm, jgraph, jvm98, and raytracer.](image-url)
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.