Escape Analysis

Florian Kübler

What is Escape Analysis?

- Escape analysis tries to **statically** reason about the **dynamic scope** of an **object**.

Traditional Use Cases

- Stack-allocations
- Scalar-replacement
- Unnecessary synchronization removal
- Identify pure methods (see purity analysis slides)
- Identify immutable data-structures
- Generate more precise call graphs

Stack Allocation

```java
public void foo() {
    Object o = new Object();
    processObject(o);
}

public void processObject(Object o) {
    // ...
}
```

- The object o can be allocated on the stack instead of the heap.
- No garbage collection is needed for o.

Scalar Replacement

```
class Circle {
    double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
}
```

```
public double foo() {
    Circle c = new Circle(1);
    return c.area();
}
```

- No object needs to be created for c.
- The call to c.area() can be inlined.
- May also require inlining analysis.

Removing Unnecessary Synchronization

```
public class AppletViewer {
  // ...
  public URL getCodeBase() {
    Object o = new Object();
    synchronized (o) {
      // ...
    }
    return this.baseURL;
  }
}
```

Synchronization is useless on o, as only this call of getCodeBase has access to it.

This code was found in JDK 8 update 151

What is Escape Analysis (cont.)?

- Escape analysis makes a statement about the object's **lifetime** and **accessibility**
- In contrast to lifetime analysis, escape analysis is more coarse grain and binds the object's lifetime/accessibility to other entities (in particular **methods** or **threads**)

Dimensions of Escaping: Lifetime

```java
public void m() { // in thread t
    Object o = new Object();
}
```

1. Lifetime(t) ≥ Lifetime(o) ≤ Lifetime(m)

Stack Allocation: ✓ Scalar Replacement: ? Synchronization Removal: ?

2. Lifetime(t) ≥ Lifetime(o) > Lifetime(m)

Stack Allocation: ✗ Scalar Replacement: ✗ Synchronization Removal: ?

3. Lifetime(o) > Lifetime(t) ≥ Lifetime(m)

Stack Allocation: ✗ Scalar Replacement: ✗ Synchronization Removal: ✗

Dimensions of Escaping: Accessibility

```java
public void m() { // in thread t
    Object o = new Object();
}
```

1. $Access(o) = \{m, t\}$

Stack Allocation: ✓ Scalar Replacement: ✓ Synchronization Removal: ✓

2. $(\exists m' \neq m \,.\, m' \in Access(o)) \wedge (\nexists t \neq t'. t' \in Access(o))$

Stack Allocation: ? Scalar Replacement: ✗/? Synchronization Removal: ✓

3. $\exists t' \neq t \,.\, t' \in Access(o)$

Stack Allocation: ✗ Scalar Replacement: ✗ Synchronization Removal: ✗

Method Calls

```java
public void foo() {
    Object o = new Object();
    bar(o);
}

public void bar(Object o) {
    // ...
}
```

On Method calls (**c**):

- Access(o) = {foo, bar, t}
- Lifetime(o) ≤ Lifetime(foo)
- Stack Allocation: ✓
- Scalar Replacement: ✗
- Synchronization Removal: ✓

Method Returns

```
public Object foo() {
    Object o = new Object();
    return o;
}

public void bar() {
    foo();
}
```

On returns (**r**):

- Access(o) = {foo, bar, t}
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) < Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ✓

Unhandled Exceptions

```
public void foo() {
   throw new Exception();
}

public void bar() {
   foo();
}
```

On abnormal returns (**a**):

- Access(o) = {foo, bar, t}
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) < Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ✓

Mutating the State of Given Parameters

```
public class MyClass {
    Object f;

    public void foo(MyClass p) {
        p.f = new Object();
    }

    public void bar() {
        foo(new MyClass);
    }
}
```

Parameter field writes(**p**):

- Access(o) = {foo, bar, t}
- Lifetime(o) > Lifetime(foo)
- Lifetime(o) < Lifetime(t)
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ?

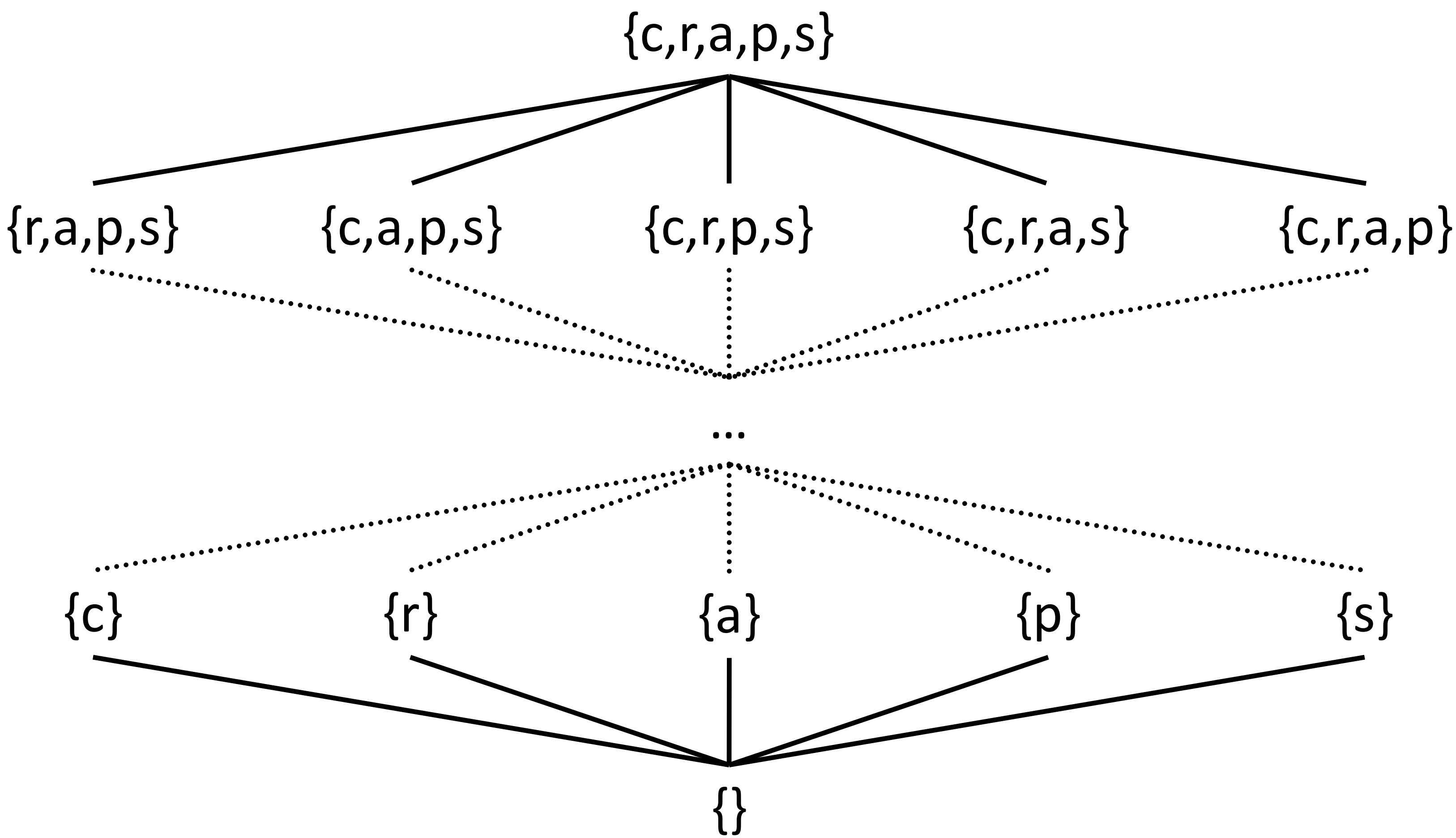
Writing Static Fields

```
public class MyClass {
  public static Object f;

  public void foo() {
    Object o = new Object();
    MyClass.f = o;
  }

  public void bar() {
    foo();
  }
}
```

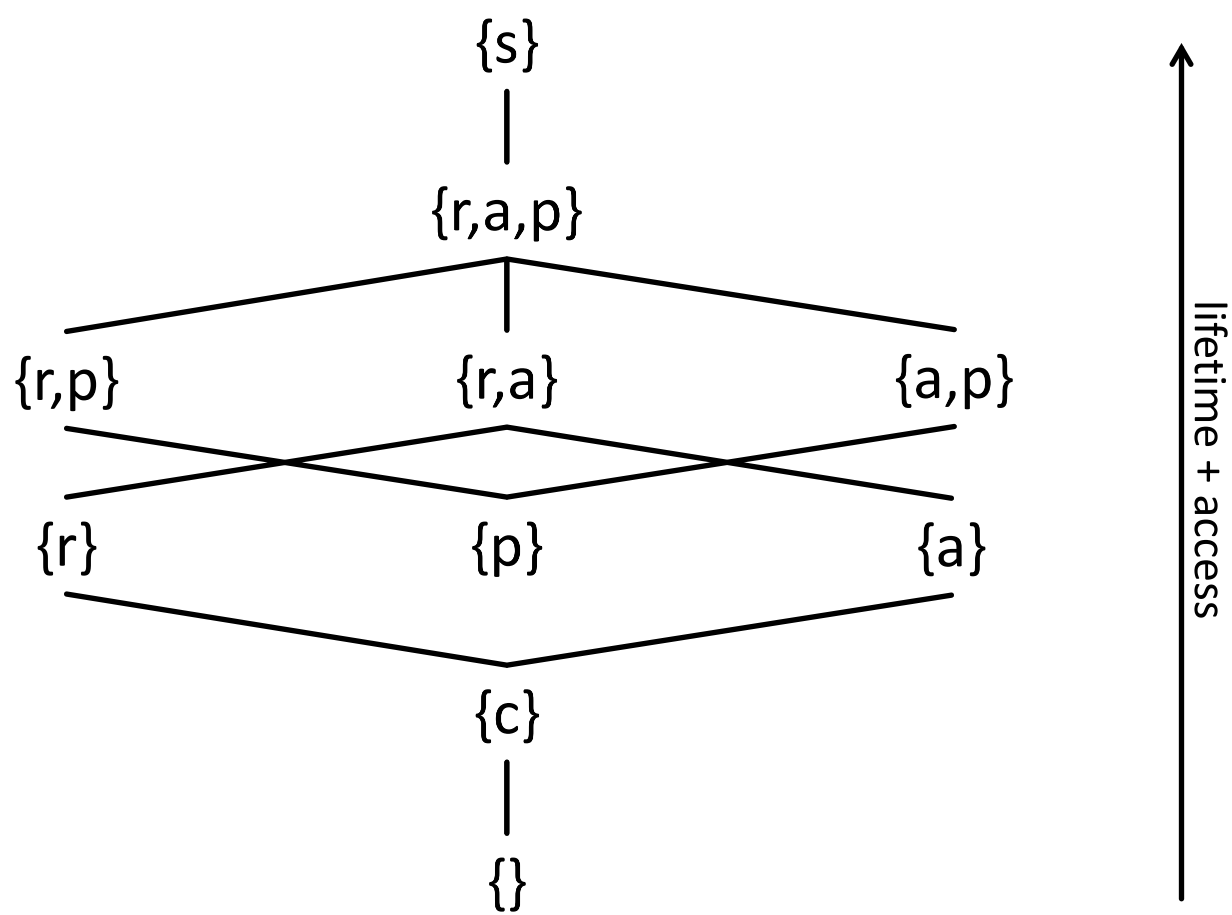
Static field writes(**s**):

- $\forall m \,.\, m \in Access(o)$
- $\forall t' \,.\, t' \in Access(o)$
- $Lifetime(o) > Lifetime(foo)$
- $Lifetime(o) > Lifetime(t)$
- Stack Allocation: ✗
- Scalar Replacement: ✗
- Synchronization Removal: ✗

Naïve Subset Lattice



Simplified Lattice



Handling Constructor Calls

```
Object o = new Object();
```

```
NEW(java.lang.Object)
DUP
INVOKESPECIAL(java.lang.Object{ void <init>() })
```

- As known from the lecture, the JVM distinguishes between object allocation and invocations of <init>.
- Distinguish between constructor and other calls as <init> does not let escape the object per-se.
- Requires (at least some) inter-procedural analysis if it is not the constructor of java.lang.Object

May-Alias Analysis and Fields

```
public class MyClass {
    Object f;
    public static Object global;

    public void foo() {
        MyClass c1 = new MyClass();
        MyClass.global = c1; // c1 escapes
        MyClass c2 = c1; // requires may-alias detection
        Object o = new Object();
        c2.f = o; // c2 does escape, therefor o does as well
    }
}
```

Research Directions

- Use escape information for more precise call graphs for libraries.
- Modularization and soundness of call-graph and points-to analysis
- For lab, seminar or thesis topics as well as for HiWi positions contact me (kuebler@cs.tu-darmstadt.de).

References

- Kotzmann, Thomas, and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization." *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005.
- Choi, Jong-Deok, et al. "Escape analysis for Java." *Acm Sigplan Notices* 34.10 (1999): 1-19.
- Kotzmann, Thomas, and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization." *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005.
- https://shipilev.net/jvm/anatomy-quarks/19-lock-elision/
- https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html
- https://dzone.com/articles/overview-of-javas-escape-analysis
- https://dzone.com/articles/escape-analysis