Program Analysis for Managed Runtimes in Presence of Dynamic Features

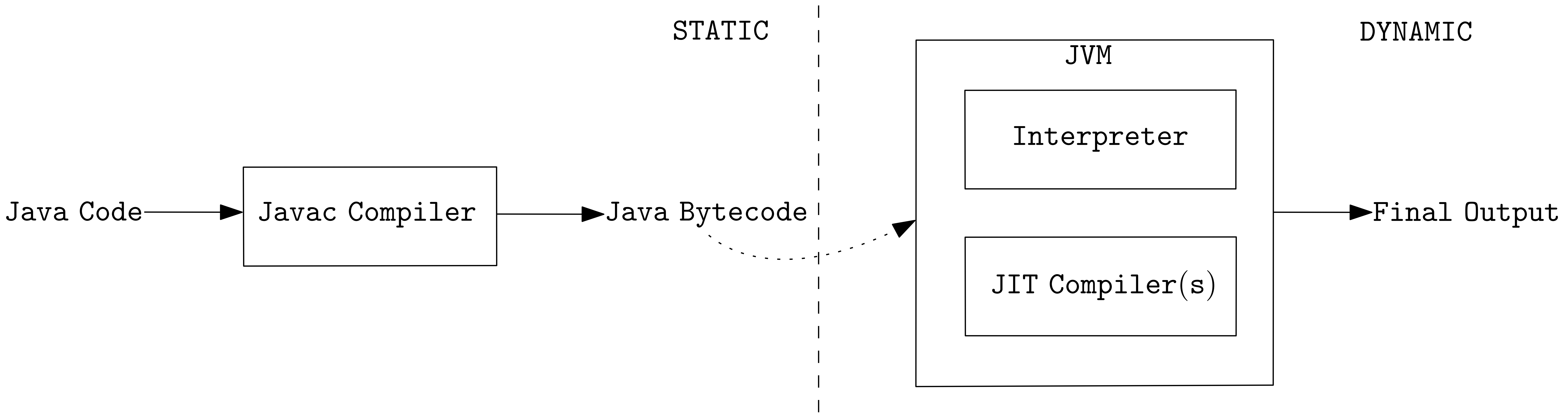
Innovations in Compiler Technology (IICT-24)

Aditya Anand

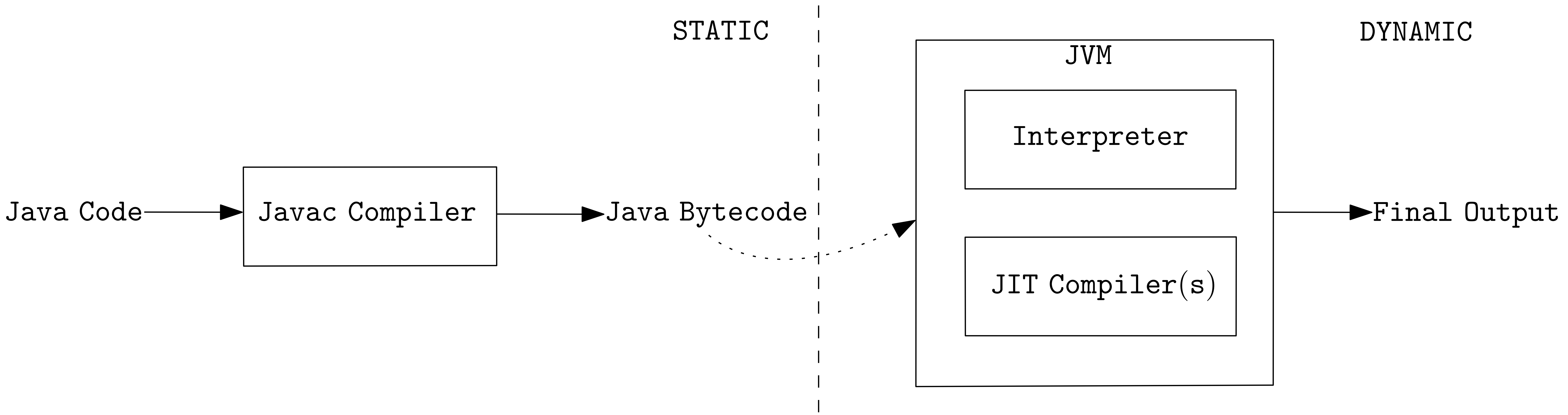
Advisor: Prof. Manas Thakur
Indian Institute of Technology Bombay

29th September 2024

Program Translation in Java

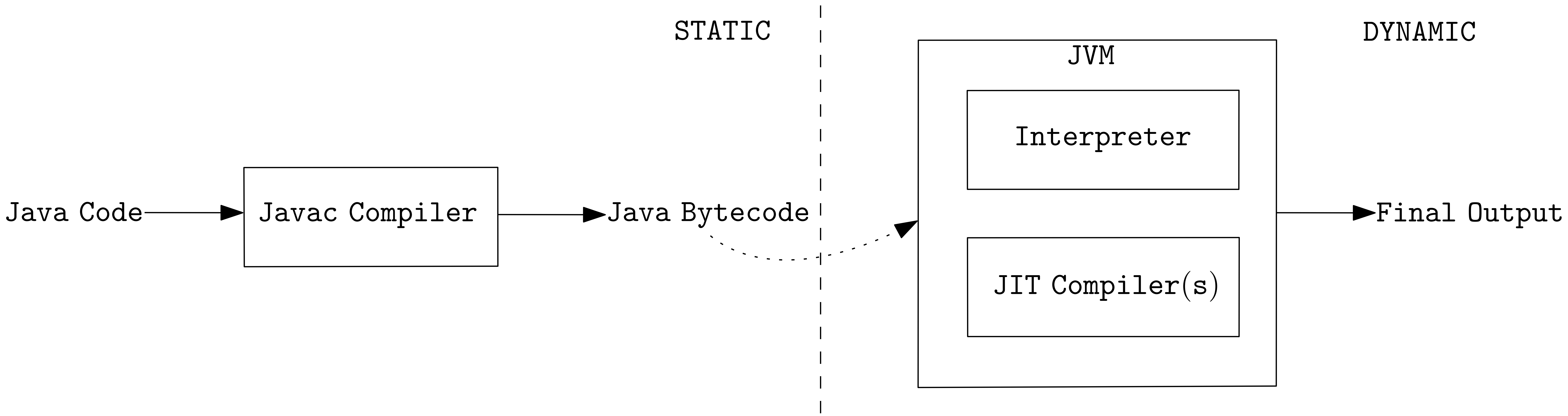


Program Translation in Java



- Static: Javac generates bytecode.

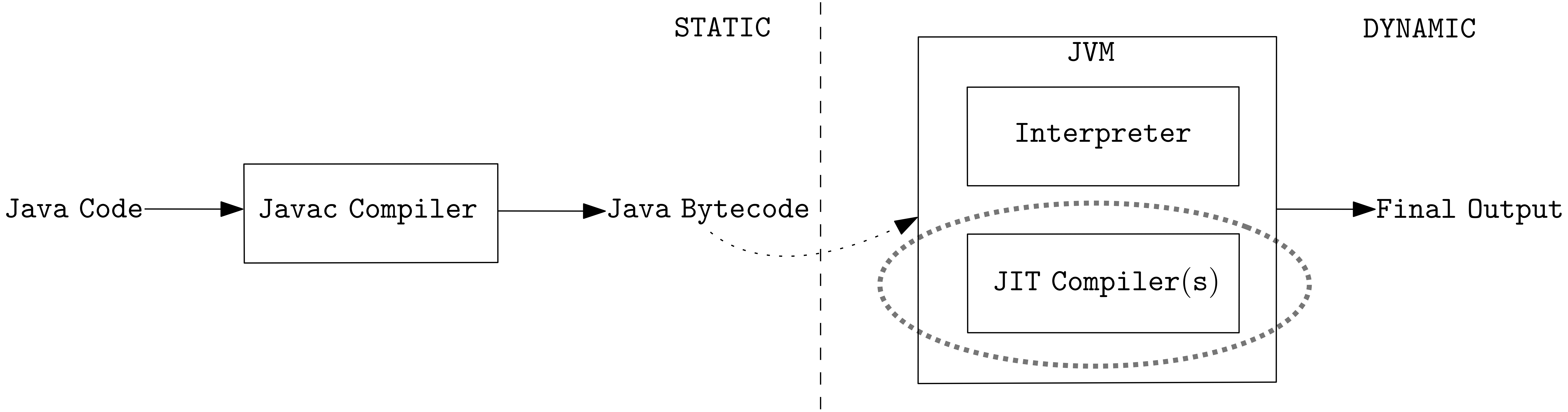
Program Translation in Java



- Static: Javac generates bytecode.

- Dynamic: Interpreter and JIT compiler generate the final output.

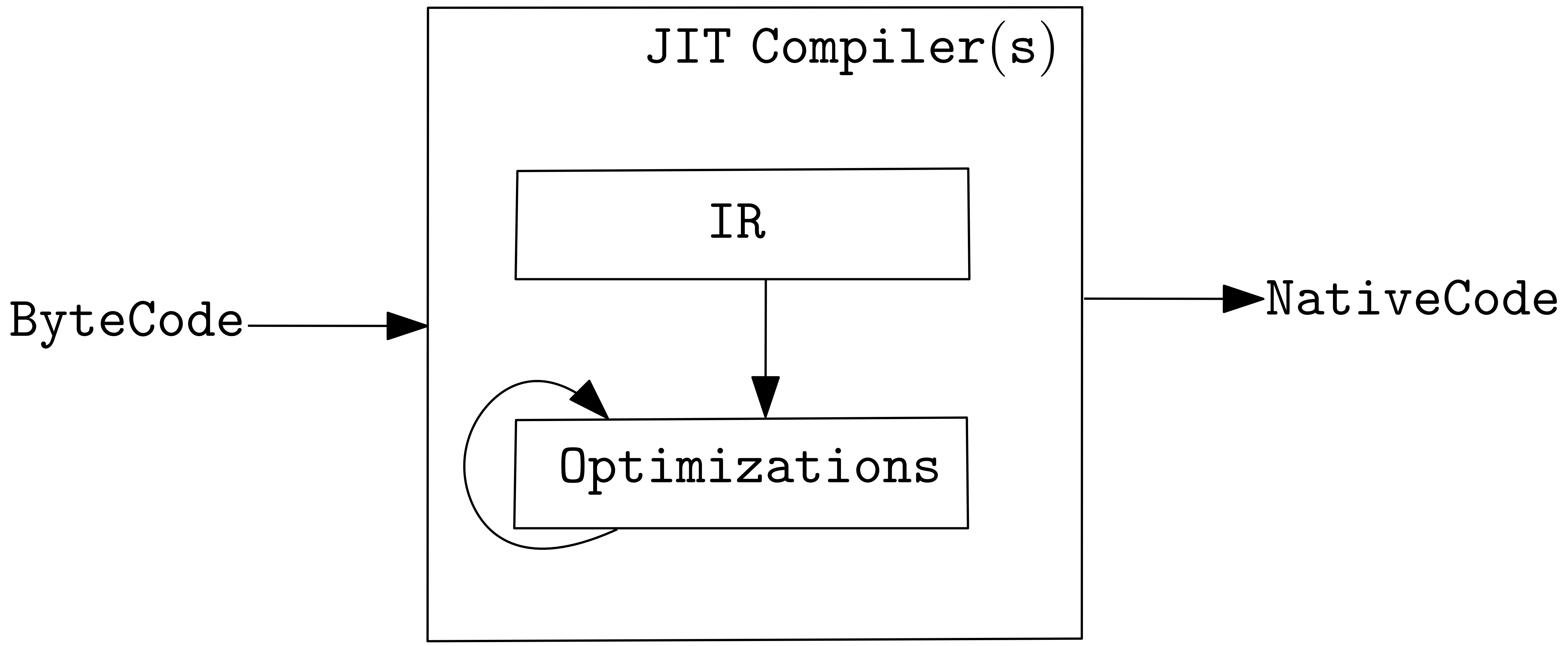
Program Translation in Java



- Static: Javac generates bytecode.

- Dynamic: Interpreter and JIT compiler generate the final output.

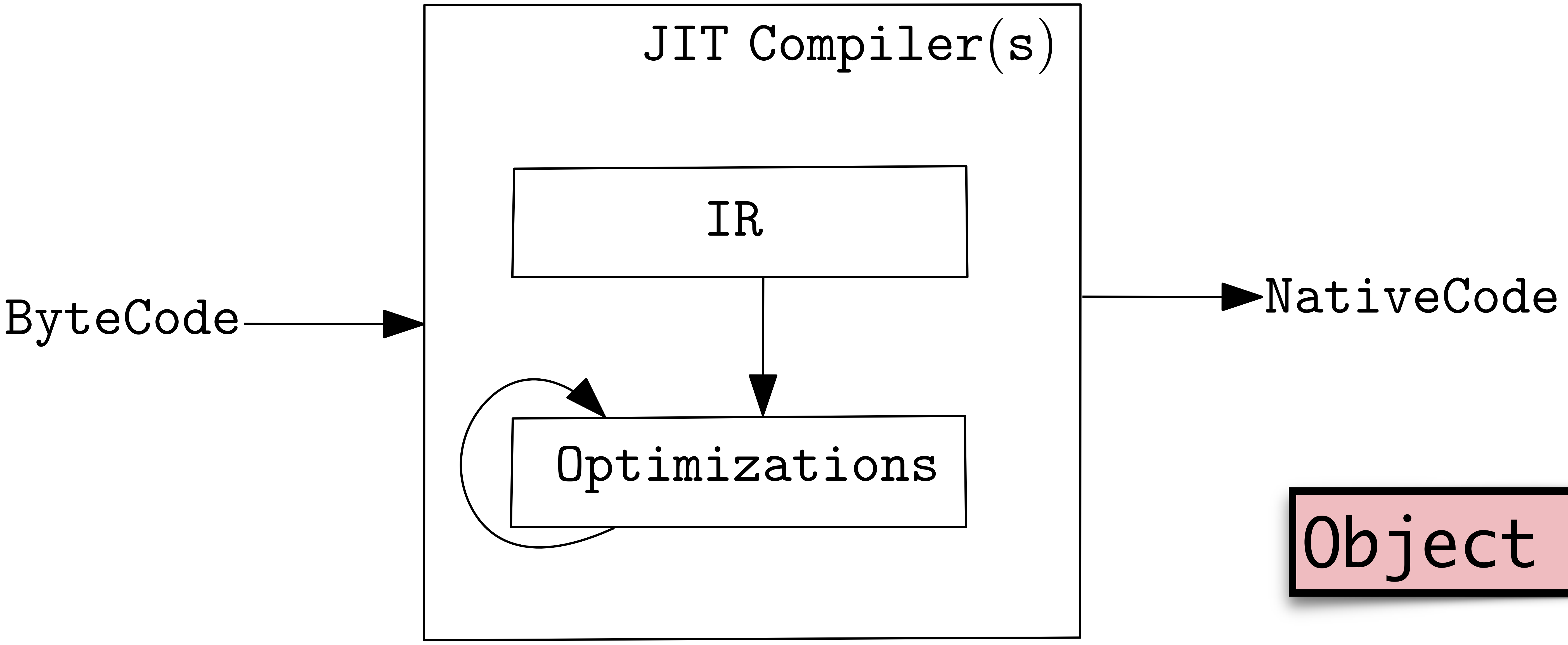
Program Translation in Java



- Static: Javac generates bytecode.

- Dynamic: Interpreter and JIT compiler generate the final output.

Program Translation in Java



Object Allocation

- Static: Javac generates bytecode.

- Dynamic: Interpreter and JIT compiler generate the final output.

Object Management in Java

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:
    - Unburden programmer from making complex allocation-deallocation decisions and reduce the possibility of harmful memory bugs.

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:
    - Unburden programmer from making complex allocation-deallocation decisions and reduce the possibility of harmful memory bugs.
- Challenges:

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:
    - Unburden programmer from making complex allocation-deallocation decisions and reduce the possibility of harmful memory bugs.
- Challenges:
    - Access time is high. — Indirections

Object Management in Java

- Managed runtime for Java allocates all objects on the heap.
- Unused objects automatically freed up by garbage collector.

```
• A a = new A(); // On heap
```

- Benefits:
  - Unburden programmer from making complex allocation-deallocation decisions and reduce the possibility of harmful memory bugs.

- Challenges:
  - Access time is high. — Indirections
  - Garbage collection is an overhead.

Stack Allocation

Stack Allocation

- Memory allocated on stack:

Stack Allocation

- Memory allocated on stack:
    - Less access time.

Stack Allocation

- Memory allocated on stack:
  - Less access time.
  - Get freed up as soon as the allocating method returns.

Stack Allocation

- Memory allocated on stack:
    - Less access time.
    - Get freed up as soon as the allocating method returns.
- Escape Analysis

Stack Allocation

- Memory allocated on stack:
  - Less access time.
  - Get freed up as soon as the allocating method returns.
- Escape Analysis
  - Determine the set of objects that do not escape the allocating method.

Stack Allocation

- Memory allocated on stack:
  - Less access time.
  - Get freed up as soon as the allocating method returns.
- Escape Analysis
  - Determine the set of objects that do not escape the allocating method.
- In case of Java:

Stack Allocation

- Memory allocated on stack:
    - Less access time.
    - Get freed up as soon as the allocating method returns.
- Escape Analysis
    - Determine the set of objects that do not escape the allocating method.
- In case of Java:
    - Escape analysis is performed: Just-in-time (JIT) compilation

Stack Allocation

- Memory allocated on stack:
    - Less access time.
    - Get freed up as soon as the allocating method returns.
- Escape Analysis
    - Determine the set of objects that do not escape the allocating method.
- In case of Java:
    - Escape analysis is performed: Just-in-time (JIT) compilation — Imprecise

Stack Allocation

- Memory allocated on stack:
  - Less access time.
  - Get freed up as soon as the allocating method returns.
- Escape Analysis
  - Determine the set of objects that do not escape the allocating method.

- In case of Java:
  - Escape analysis is performed: Just-in-time (JIT) compilation — Imprecise
  - Very few objects get allocated on stack.

Static Analysis for Stack Allocation

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to $\boxed{\text{optimistically}}$ allocate objects on stack at runtime.
- Challenges:

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to $\boxed{\text{optimistically}}$ allocate objects on stack at runtime.
- Challenges:
    1. Dynamic Class Loading (DCL)

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:
    1. Dynamic Class Loading (DCL)
    2. Hot-Code Replacement (HCR)

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:
    1. Dynamic Class Loading (DCL)
    2. Hot-Code Replacement (HCR)
    3. Callbacks

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:

1. Dynamic Class Loading (DCL)
2. Hot-Code Replacement (HCR)
3. Callbacks

Dynamic Features

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:
    1. Dynamic Class Loading (DCL)
    2. Hot-Code Replacement (HCR)
    3. Callbacks
    4. Tampered Static Analysis Result

Dynamic Features

Static Analysis for Stack Allocation

- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:
    1. Dynamic Class Loading (DCL)
    2. Hot-Code Replacement (HCR)
    3. Callbacks
    4. Tampered Static Analysis Result

  Dynamic Features
- An object that was stack allocated based on static-analysis results, might start escaping at run-time.

1. Dynamic Classloading

1. Dynamic Classloading

- Loading the class on demand — **lazy loading**.

1. Dynamic Classloading

- Loading the class on demand — **lazy loading**.
- Allows loading Java code at run-time that was not known before program execution started.

1. Dynamic Classloading

- Loading the class on demand — **lazy loading**.
- Allows loading Java code at run-time that was not known before program execution started.
- Uses the **Reflection API** to achieve dynamic loading.

1. Dynamic Classloading

- Loading the class on demand — **lazy loading**.
- Allows loading Java code at run-time that was not known before program execution started.
- Uses the **Reflection API** to achieve dynamic loading.

```
public static void main(String args[]) {
 try{
     Class<?> cls = Class.forName("ClassName");
     Object obj = cls.getDeclaredConstructor().newInstance();
     Method method = cls.getMethod("MethodName");
     Method.invoke(obj);
 } catch (Exception e) {}
}
```

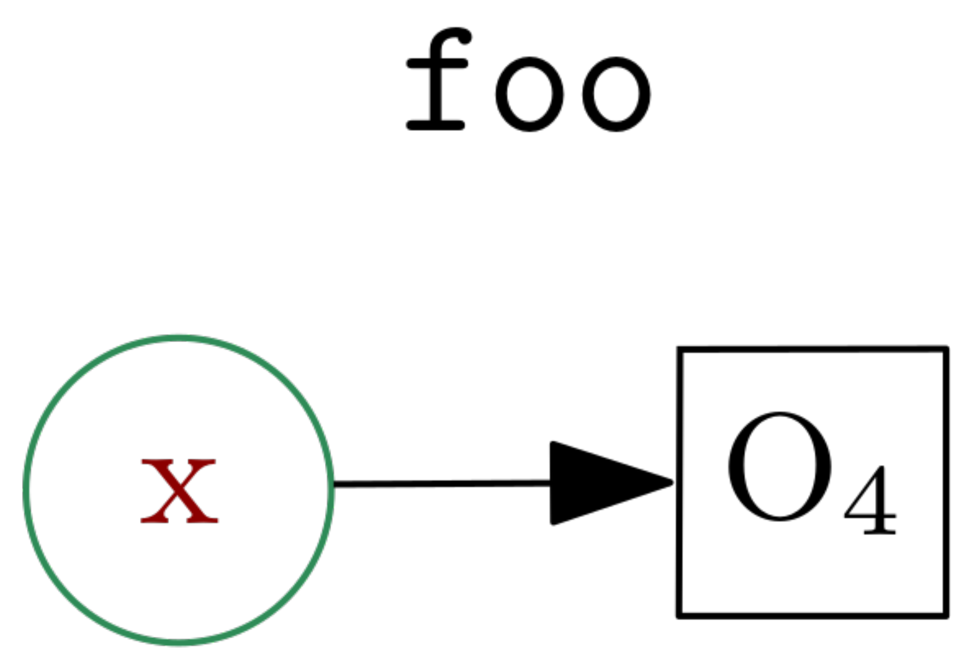
Reflection

Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();  // O4
    A y = new A();  // O5
    x.f = new A();  // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
```

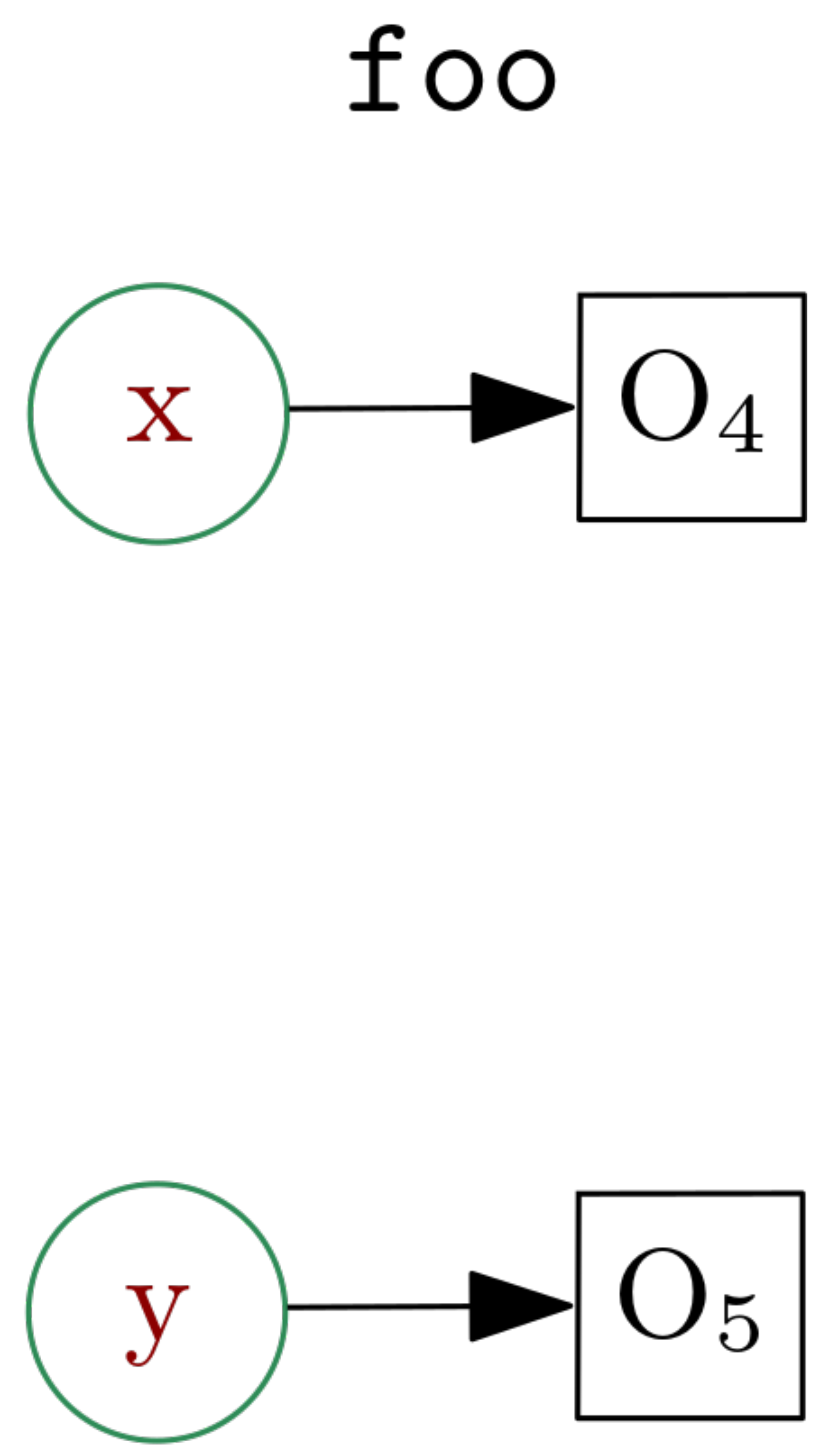
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
```



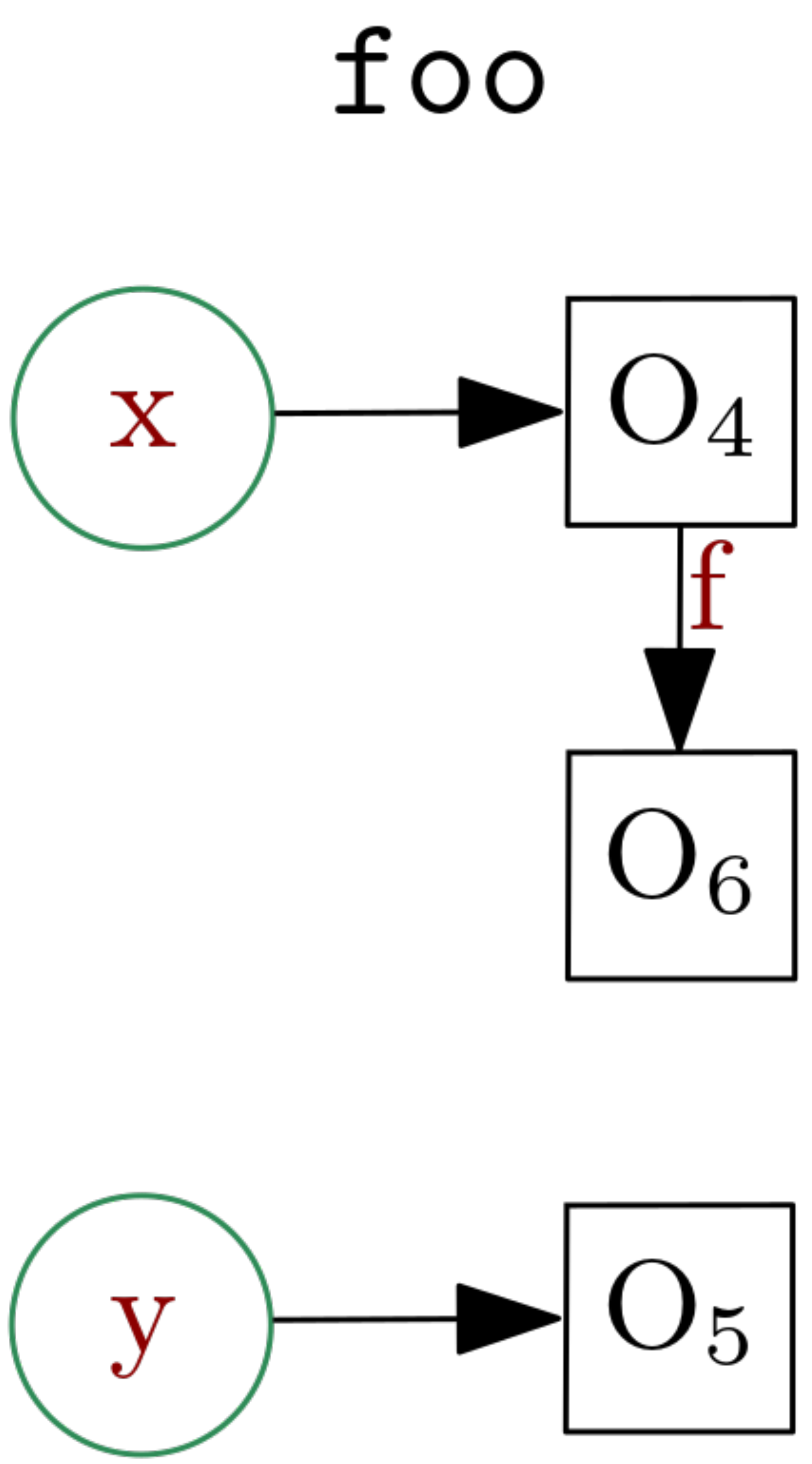
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
      A x = new A();   // O4
      A y = new A();   // O5
      x.f = new A();   // O6
      A p = x.f;
      bar(p, y);
      r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
      p1.f = p2;
  } /* method bar */
} /* class A */
```



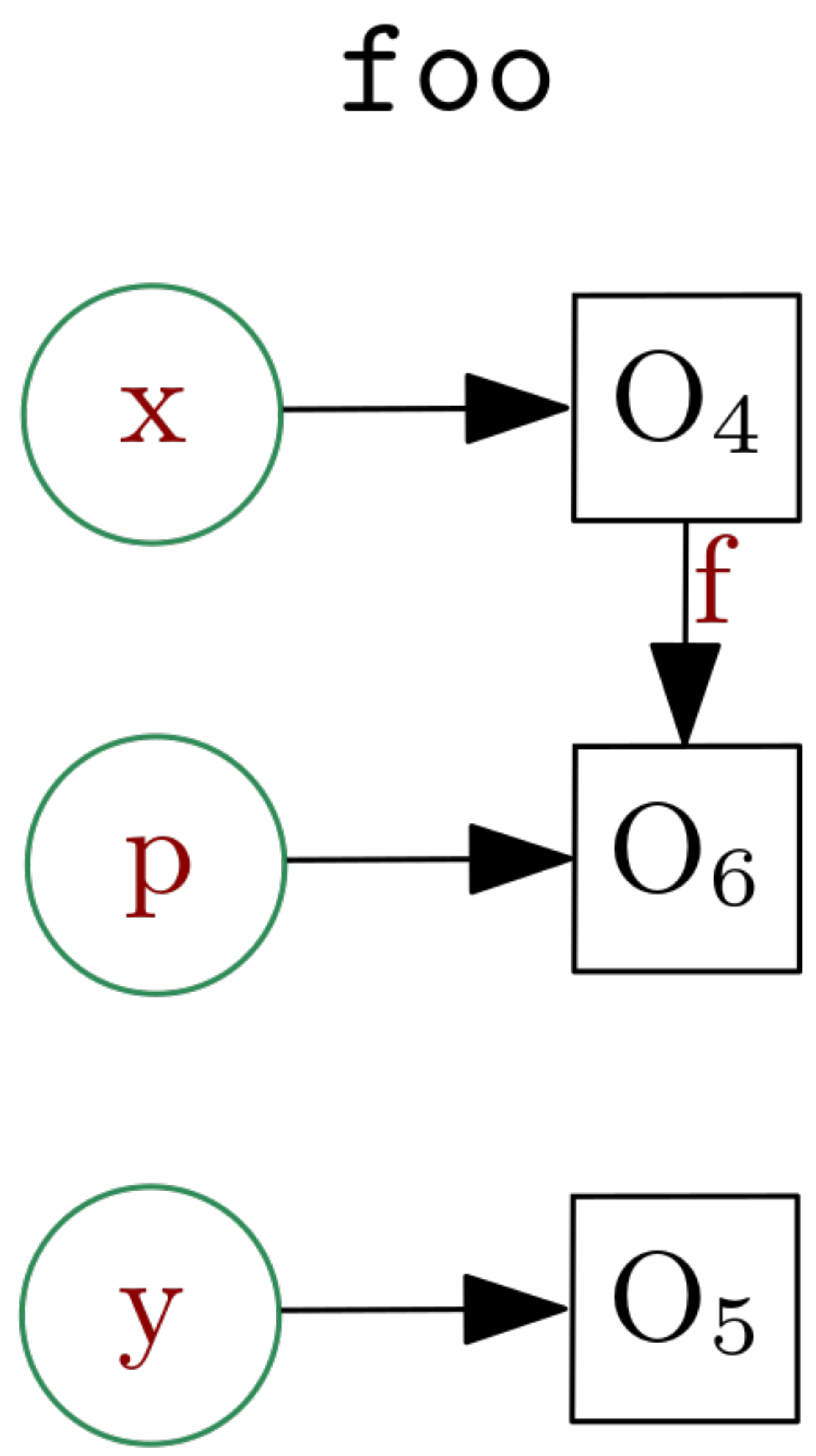
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
```



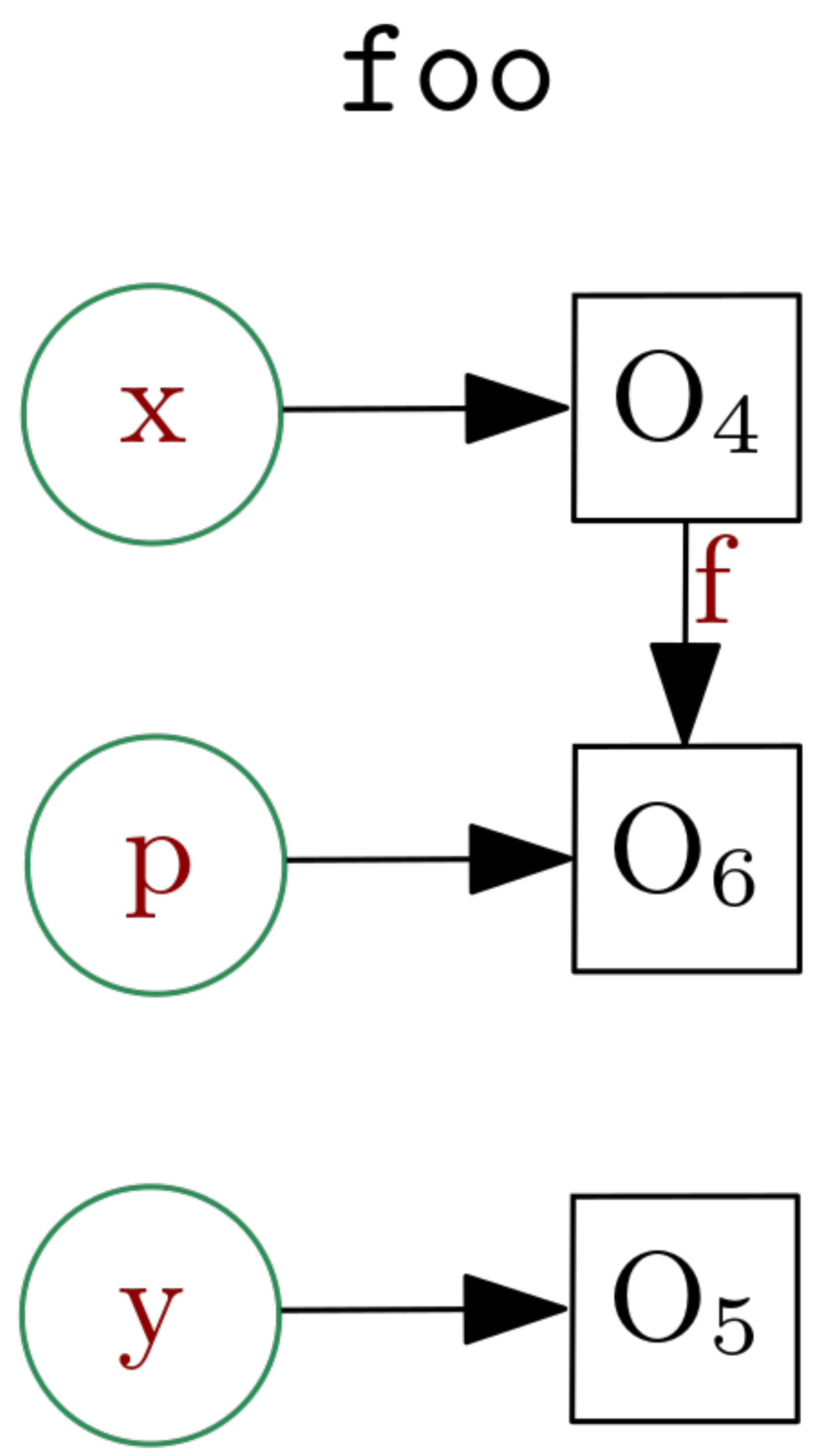
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
```



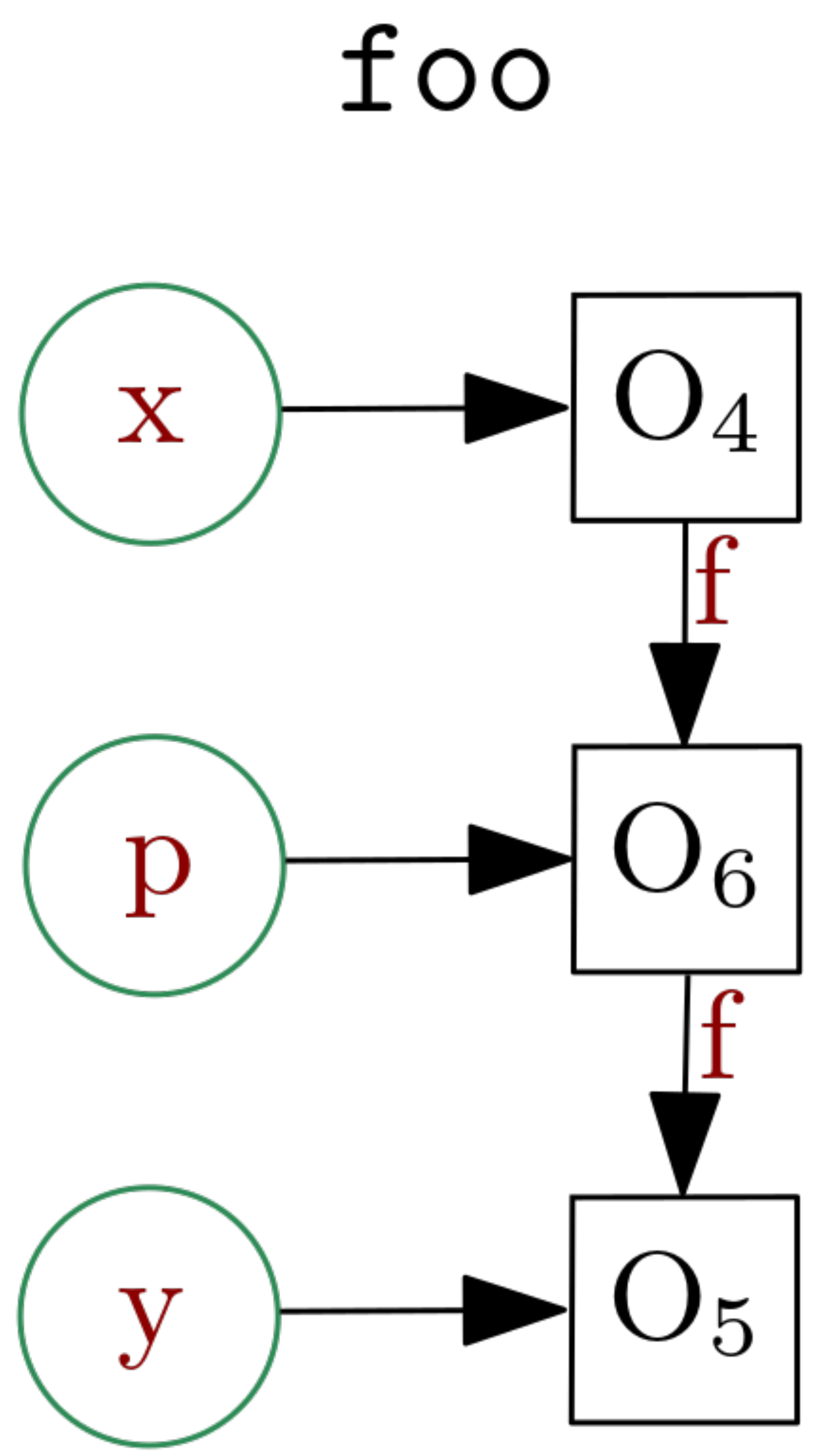
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
```



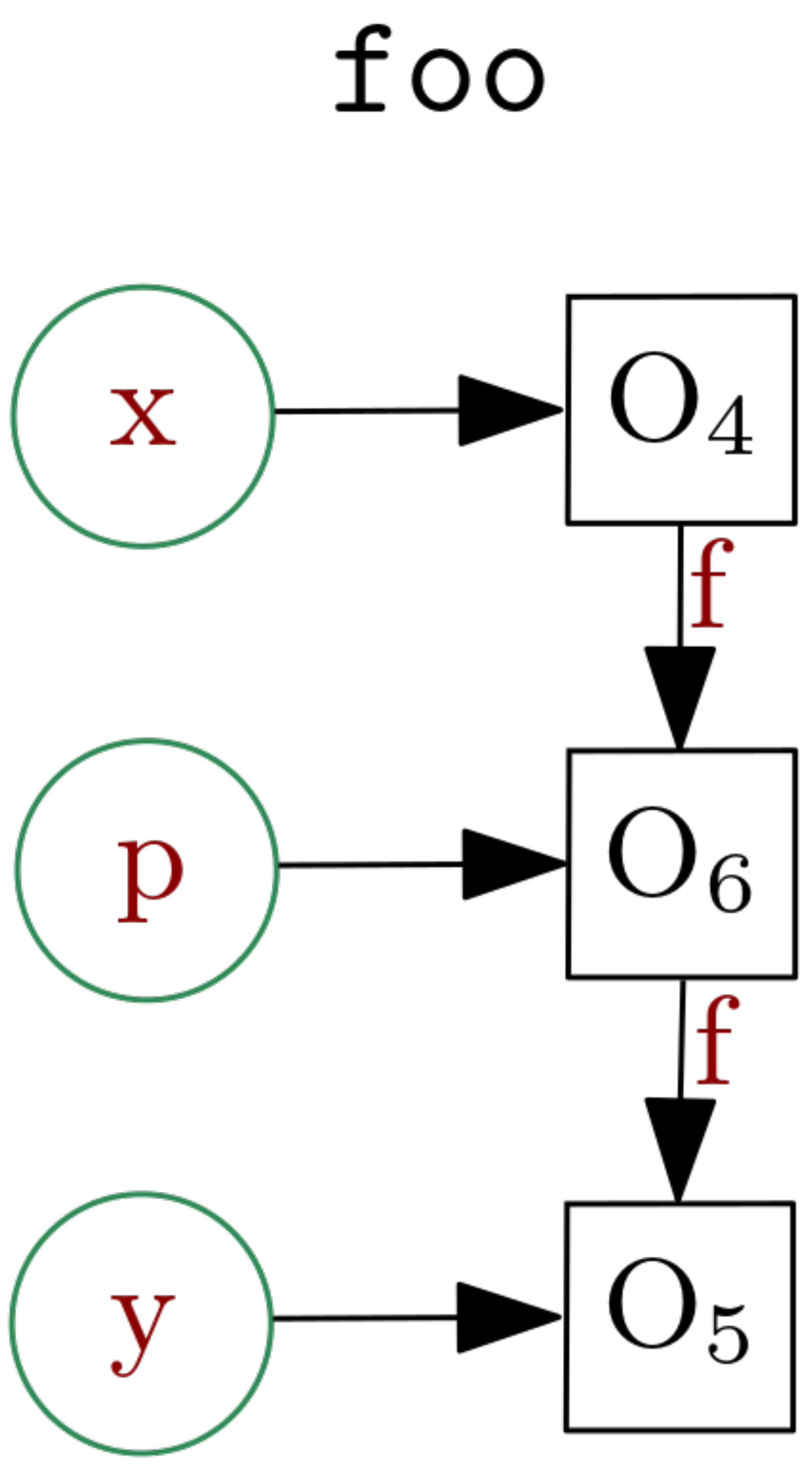
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();  // O4
     A y = new A();  // O5
     x.f = new A();  // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
```



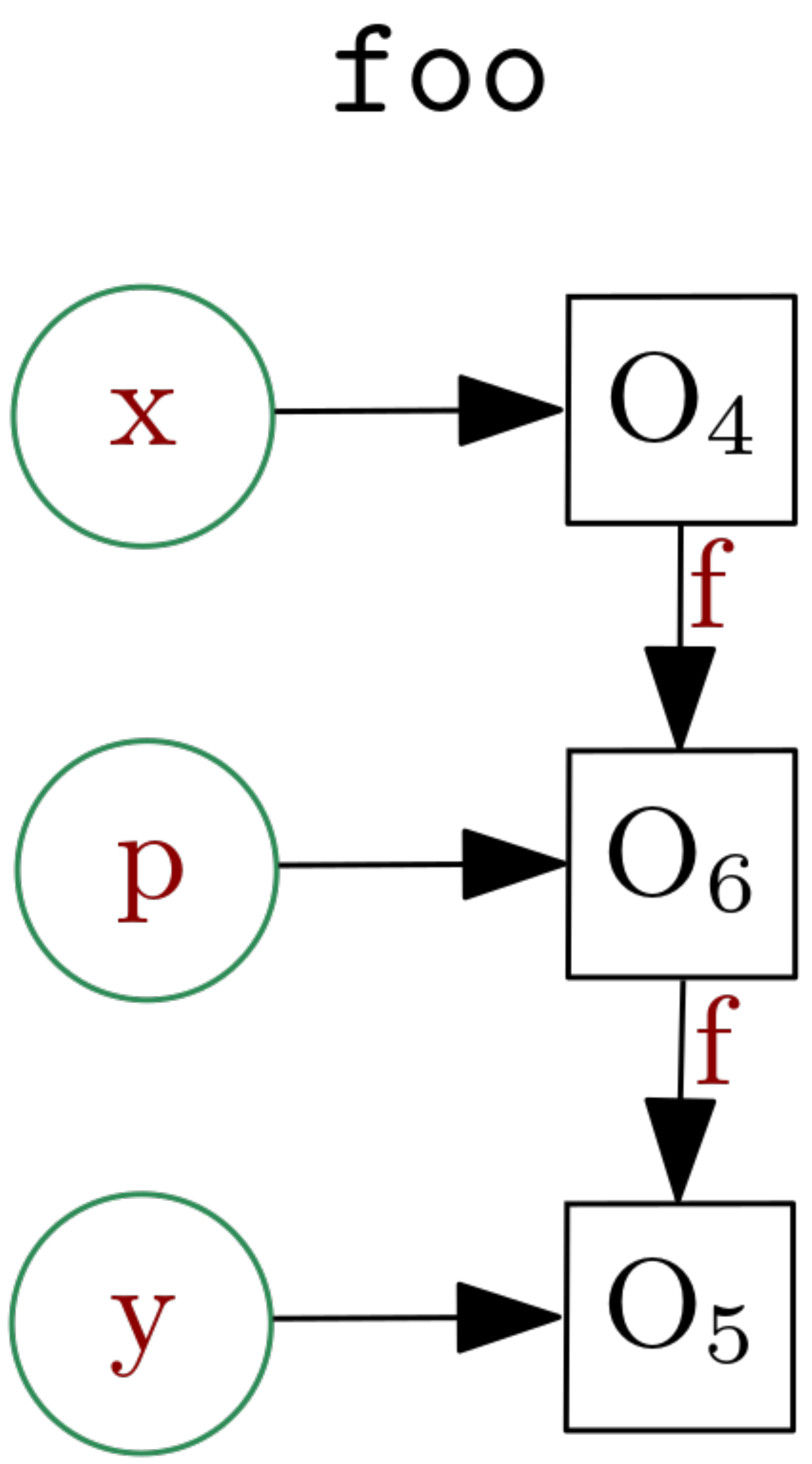
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
```



Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
```

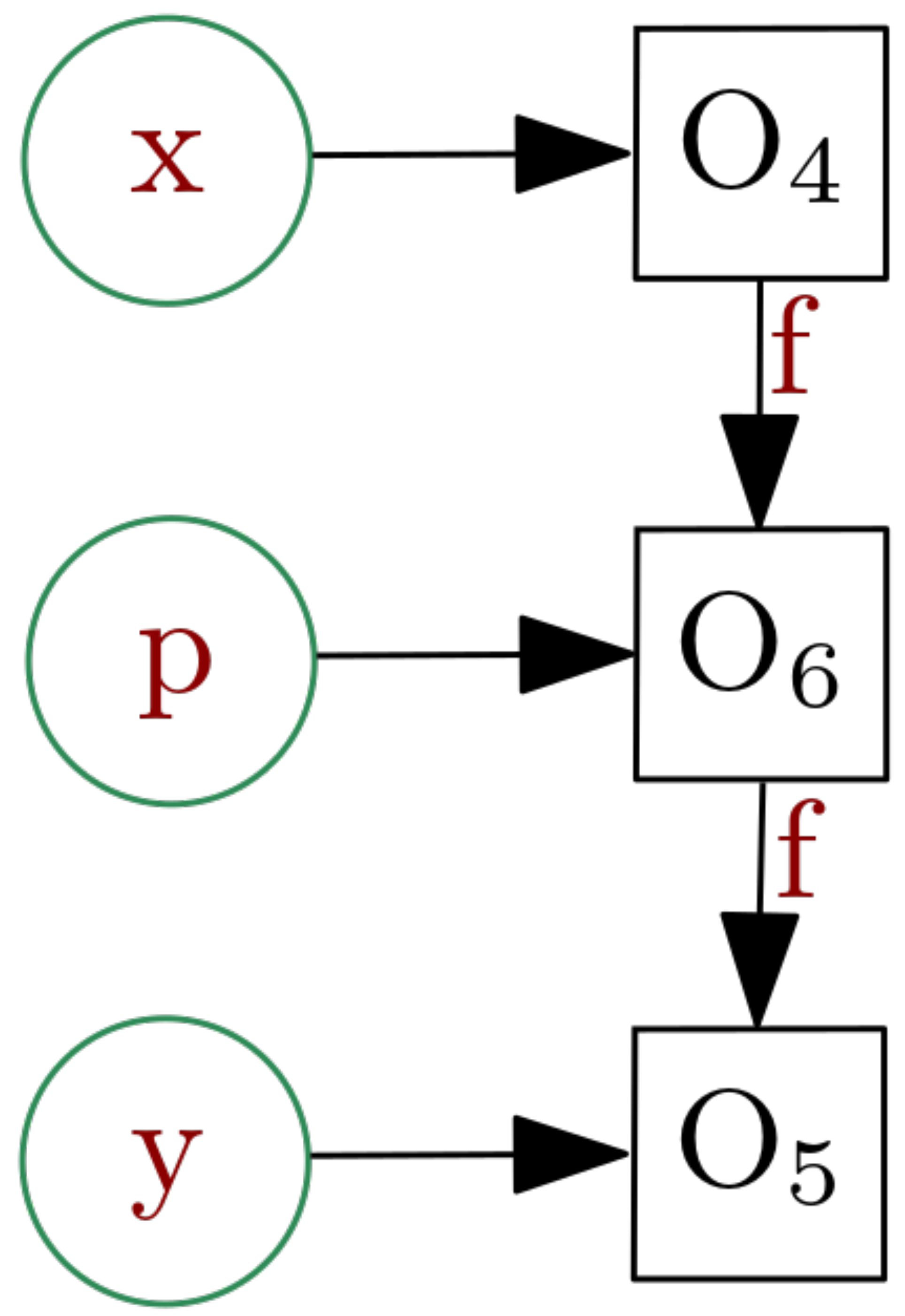


Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
class B extends A
  void zar(A p, A q) {
    q.f = p;
  } /* method zar */
} /* class B */
```

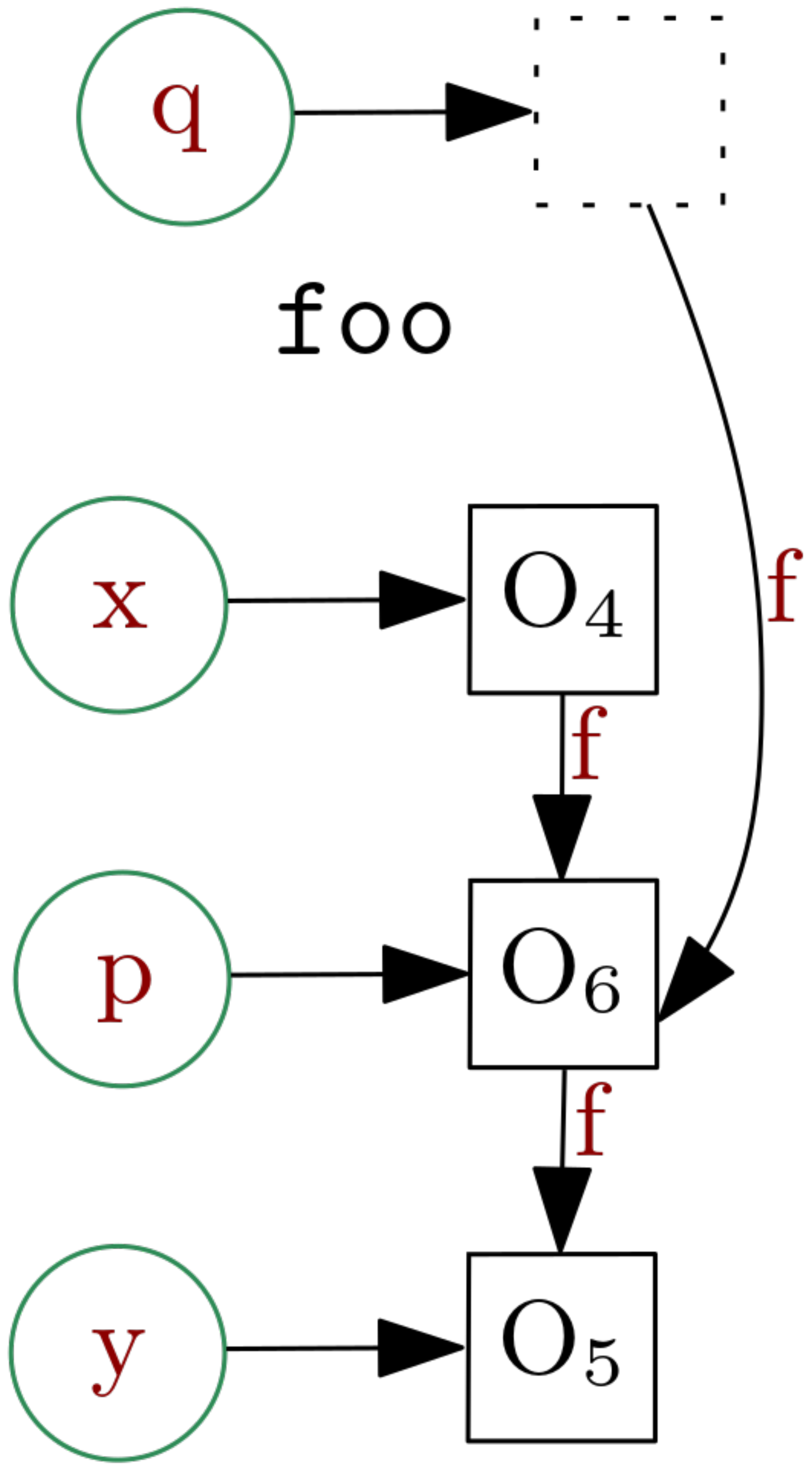
Dynamically loaded

foo



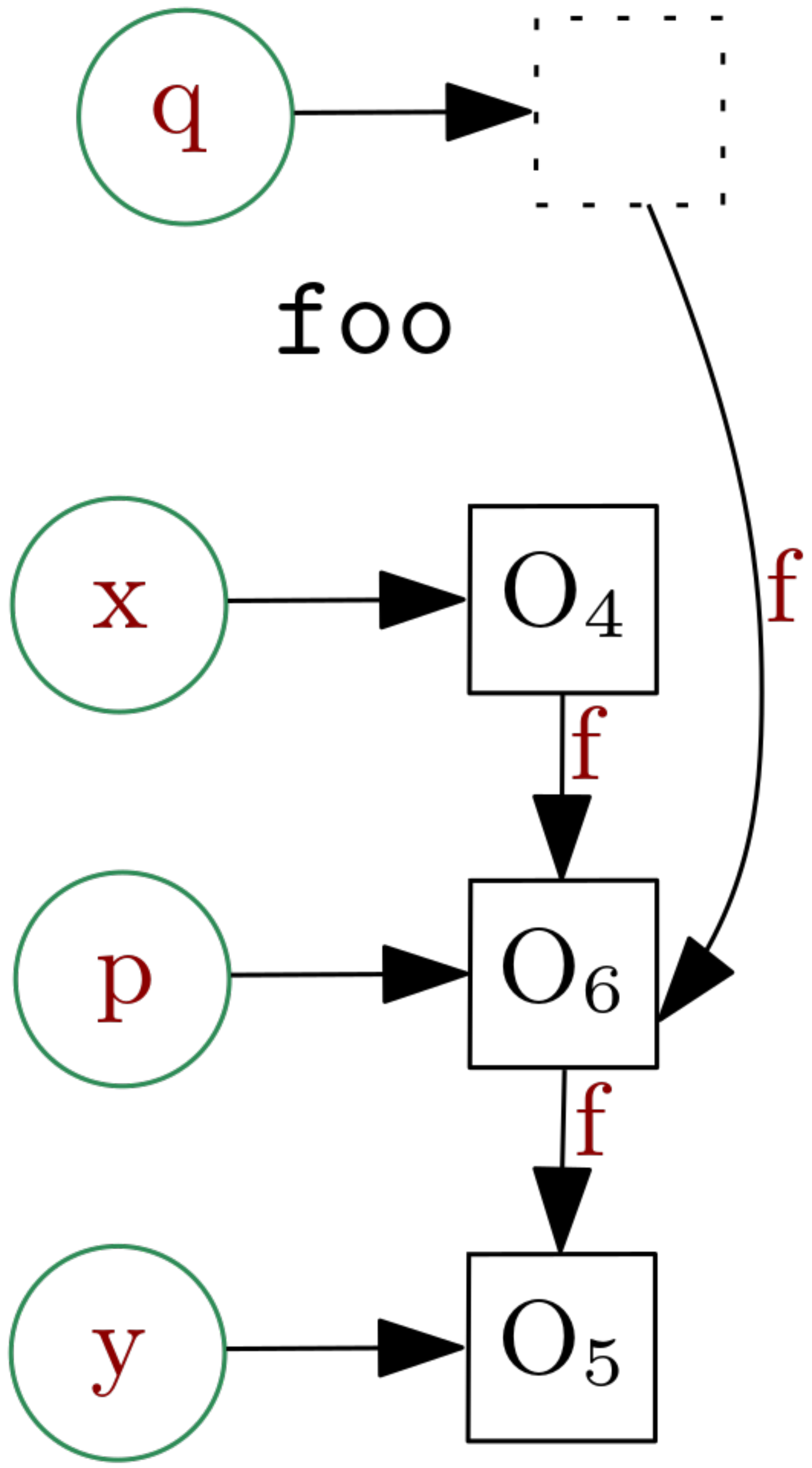
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
class B extends A
  void zar(A p, A q) {
    q.f = p;
  } /* method zar */
} /* class B */
```



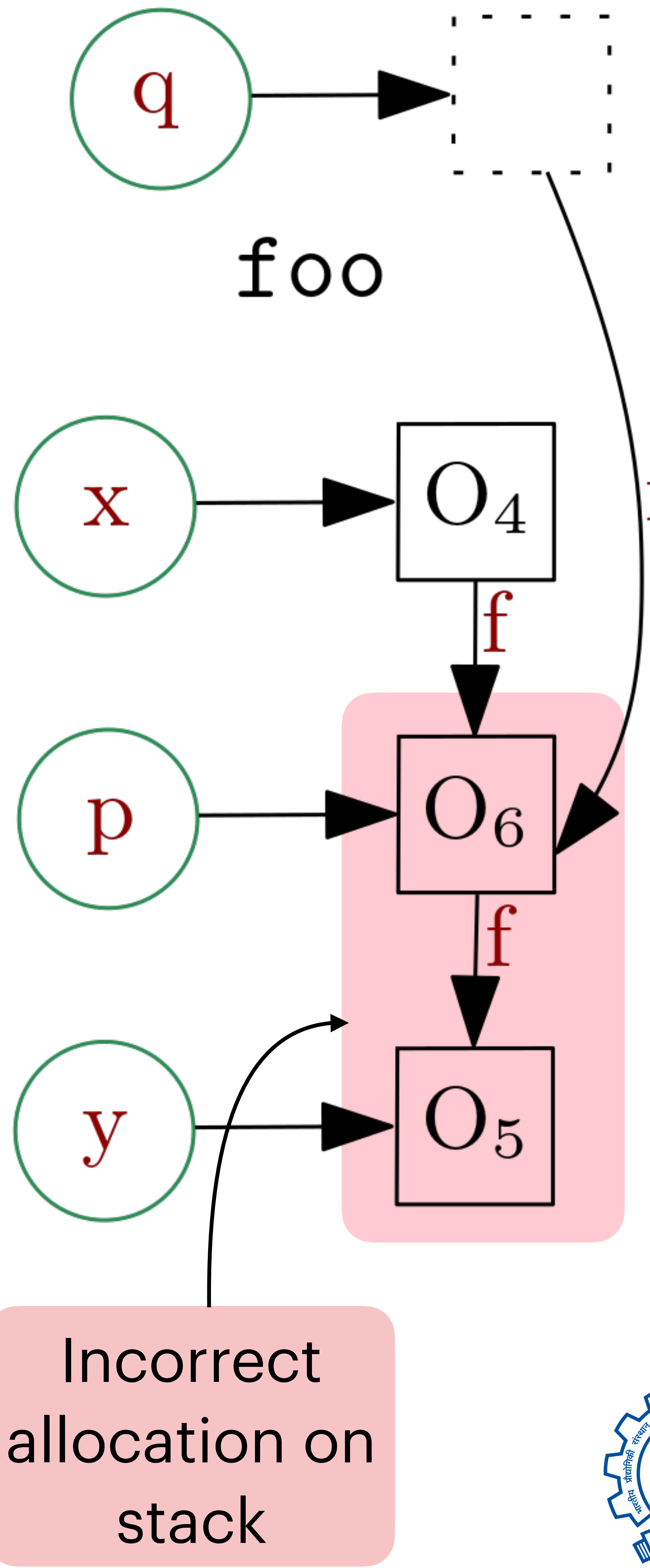
Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
class B extends A
  void zar(A p, A q) {
     q.f = p;
  } /* method zar */
} /* class B */
```



Dynamic Class Loading Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
class B extends A
  void zar(A p, A q) {
     q.f = p;
  } /* method zar */
} /* class B */
```



2. Hot Code Replacement

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
    - Mostly used during debugging — allows code to change on the fly.

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
  - Mostly used during debugging — allows code to change on the fly.
  - Most modern IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support HCR.

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
  - Mostly used during debugging — allows code to change on the fly.
  - Most modern IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support HCR.
- Possible changes are limited:

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
    - Mostly used during debugging — allows code to change on the fly.
    - Most modern IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support HCR.
- Possible changes are limited:
    - Modifying method bodies are supported.

2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
    - Mostly used during debugging — allows code to change on the fly.
    - Most modern IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support HCR.
- Possible changes are limited:
    - Modifying method bodies are supported.
    - Adding/removing methods, modifying class hierarchy, changing method signature — not allowed.

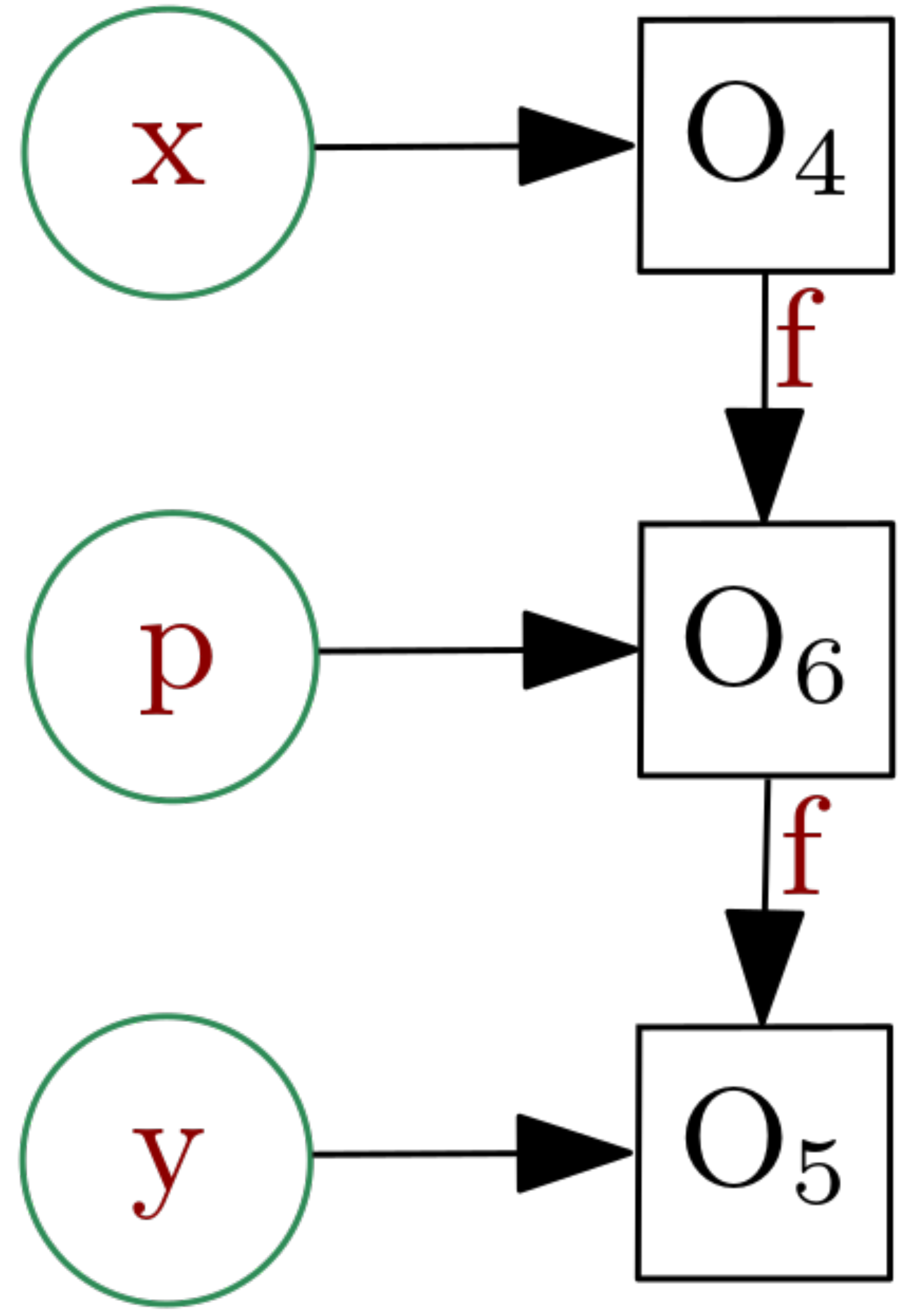
2. Hot Code Replacement

- Allows developers to change the code of a running Java application without needing to restart it.
    - Mostly used during debugging — allows code to change on the fly.
    - Most modern IDEs like Eclipse, IntelliJ IDEA, Netbeans etc support HCR.
- Possible changes are limited:
    - Modifying method bodies are supported.
    - Adding/removing methods, modifying class hierarchy, changing method signature — not allowed.
- Using static analysis while having code change at run-time may give unsound results.

HCR Example

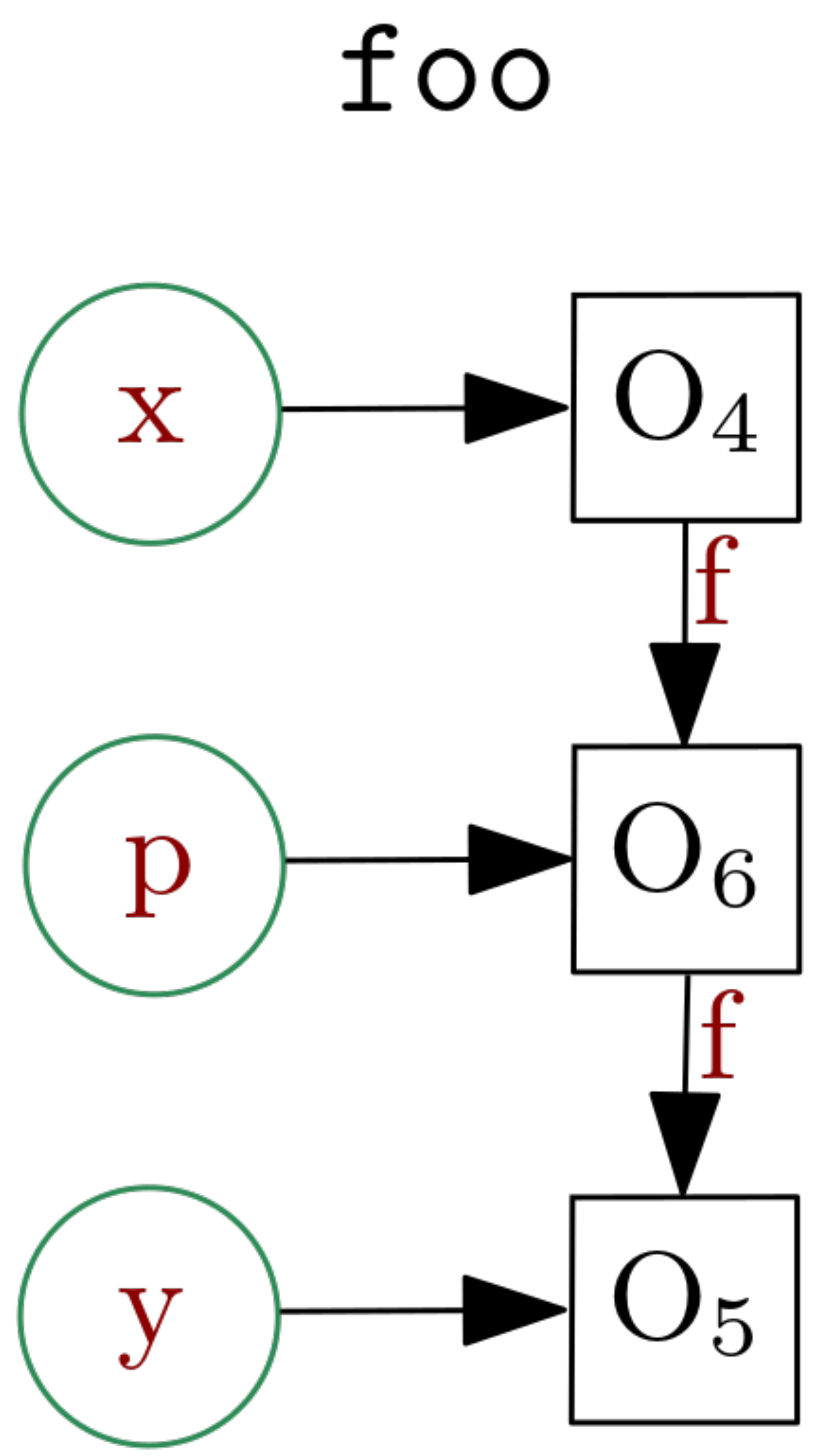
```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
    . . .
  }
} /* class A */
```

foo



HCR Example

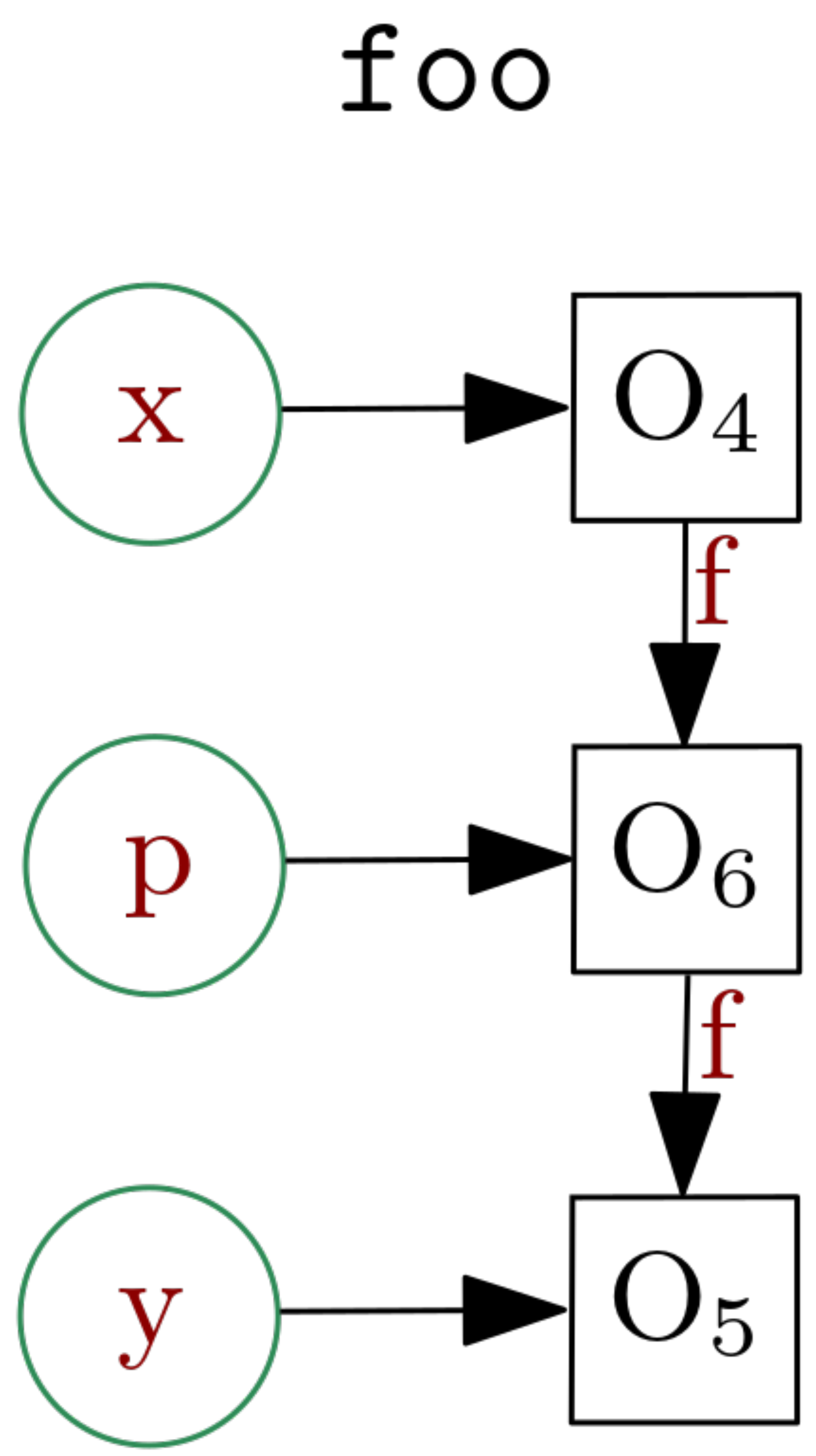
```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
       . . .
  }
} /* class A */
```



HCR Example

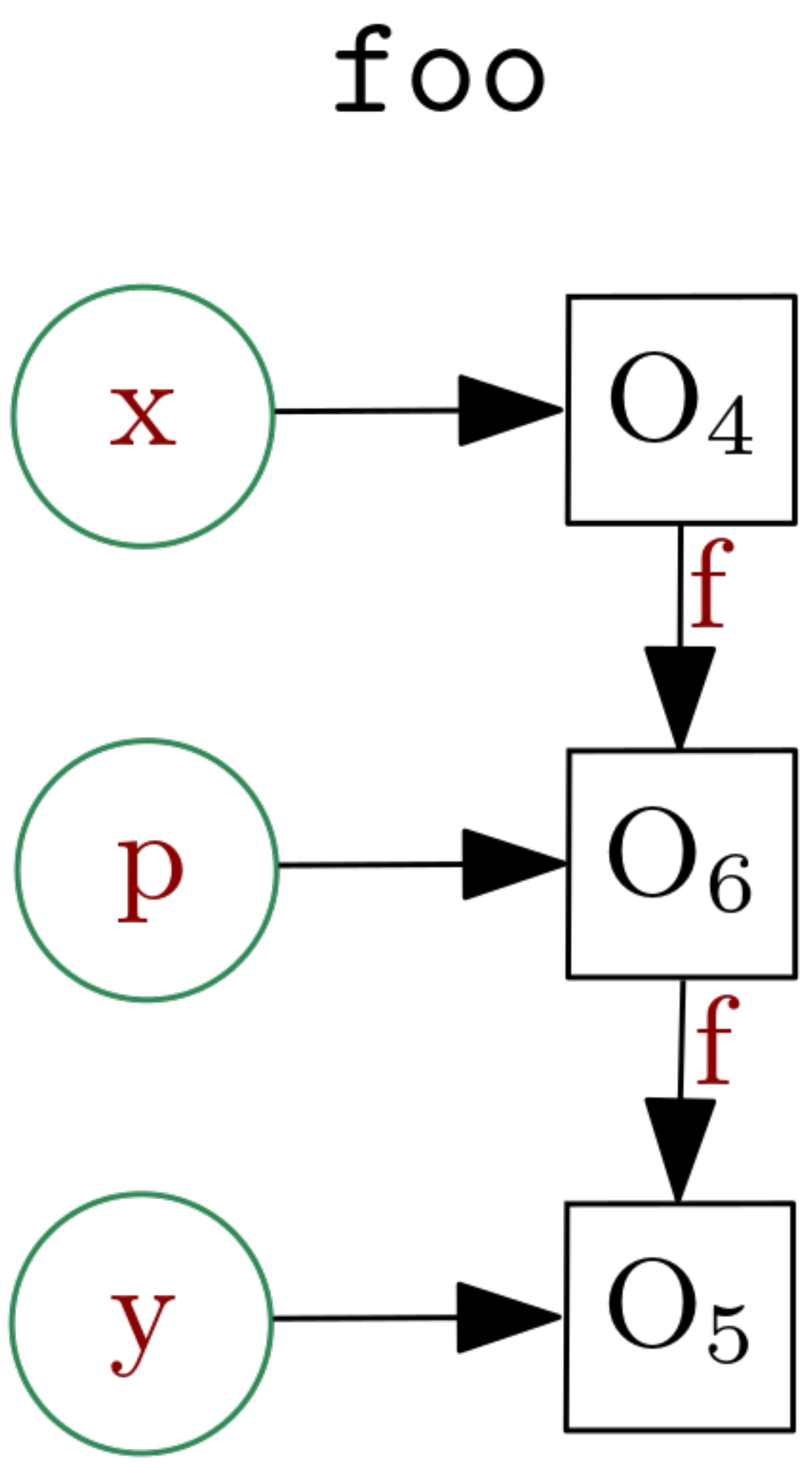
```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {

  }
} /* class A */
```



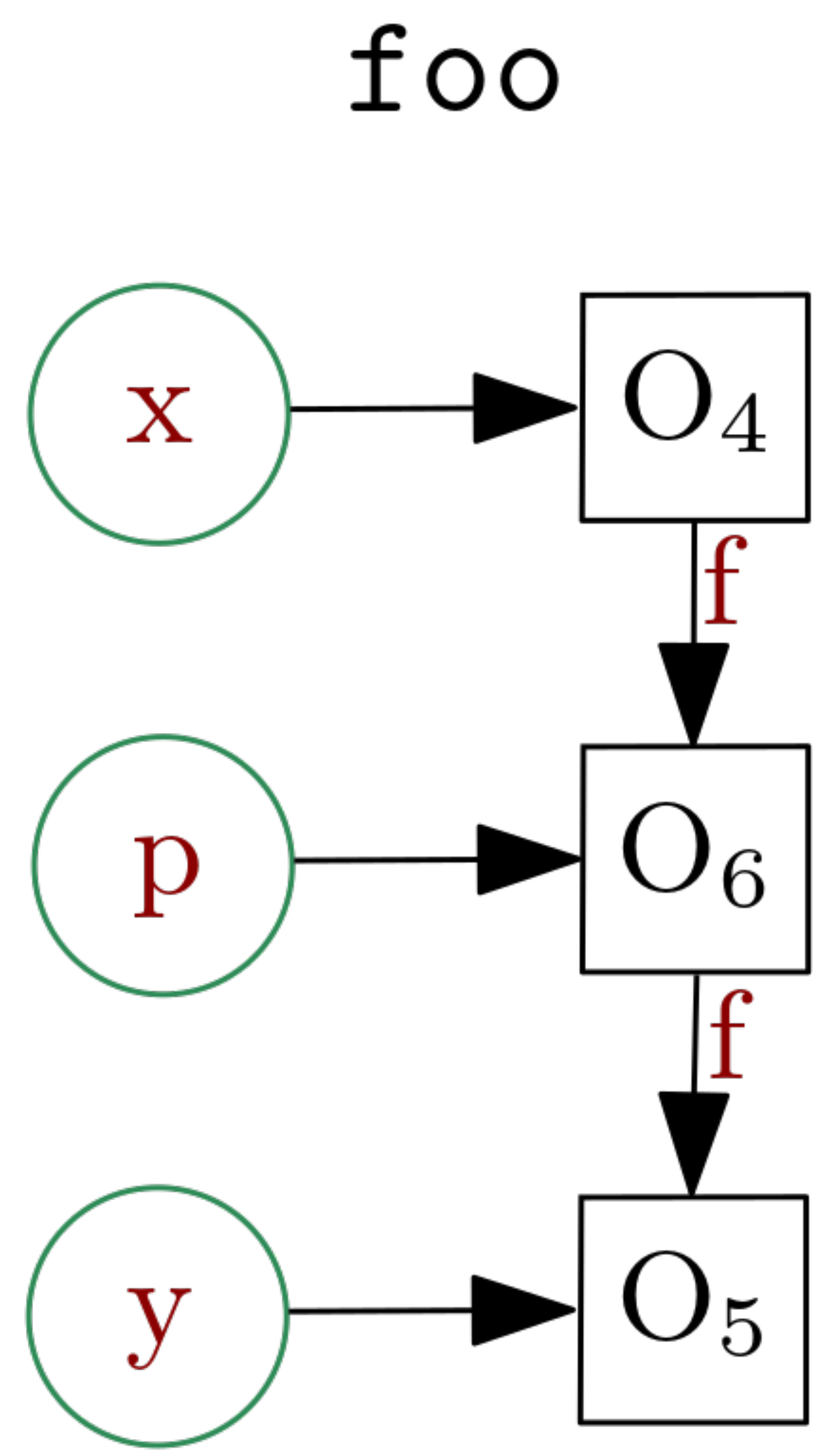
HCR Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();  // O4
     A y = new A();  // O5
     x.f = new A();  // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
     q.f = p;
  }
} /* class A */
```



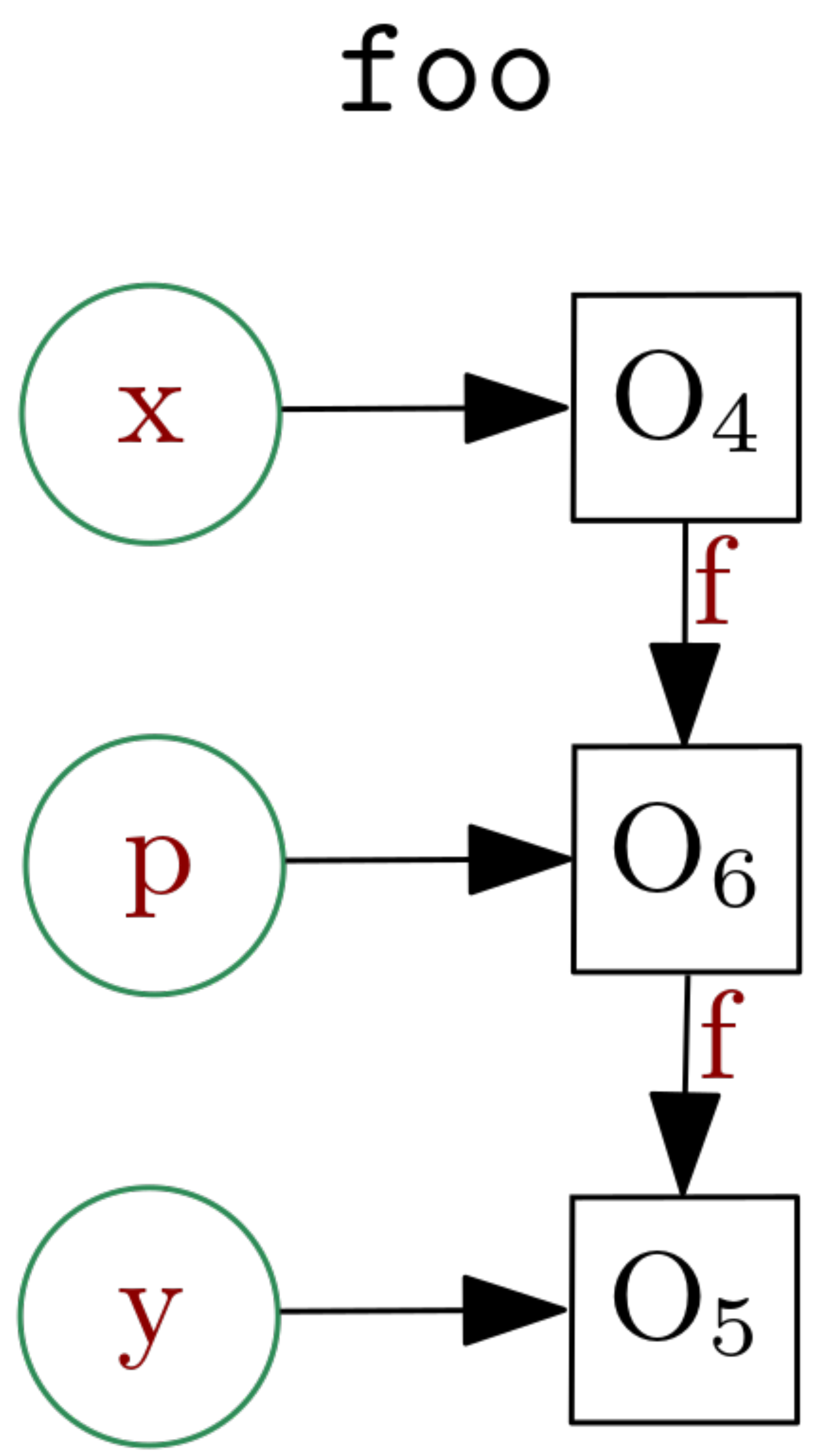
HCR Example

```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
    q.f = p;
  }
} /* class A */
```



HCR Example

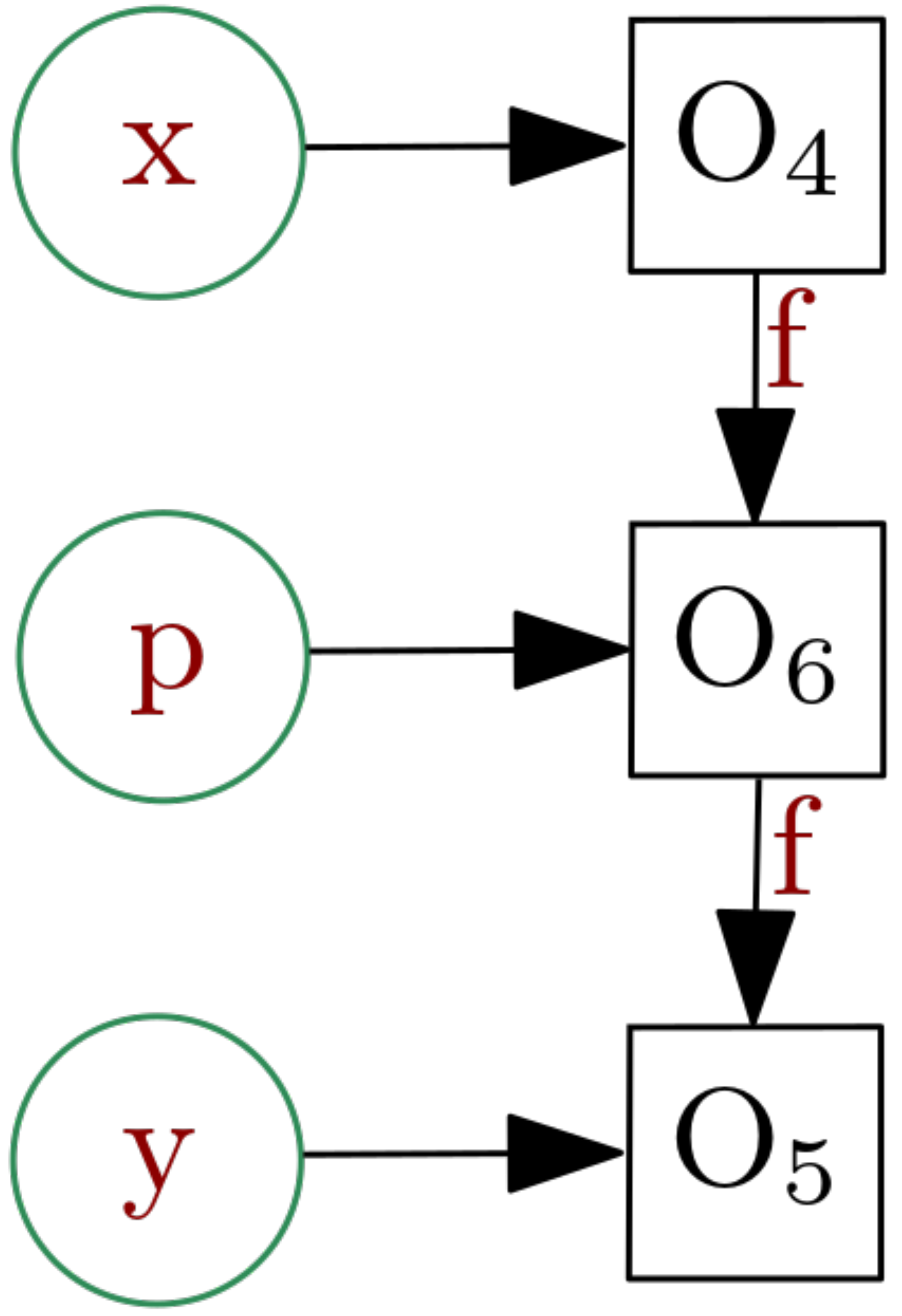
```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
     q.f = p;
  }
} /* class A */
```



HCR Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
     q.f = p;
  }
} /* class A */
```

foo

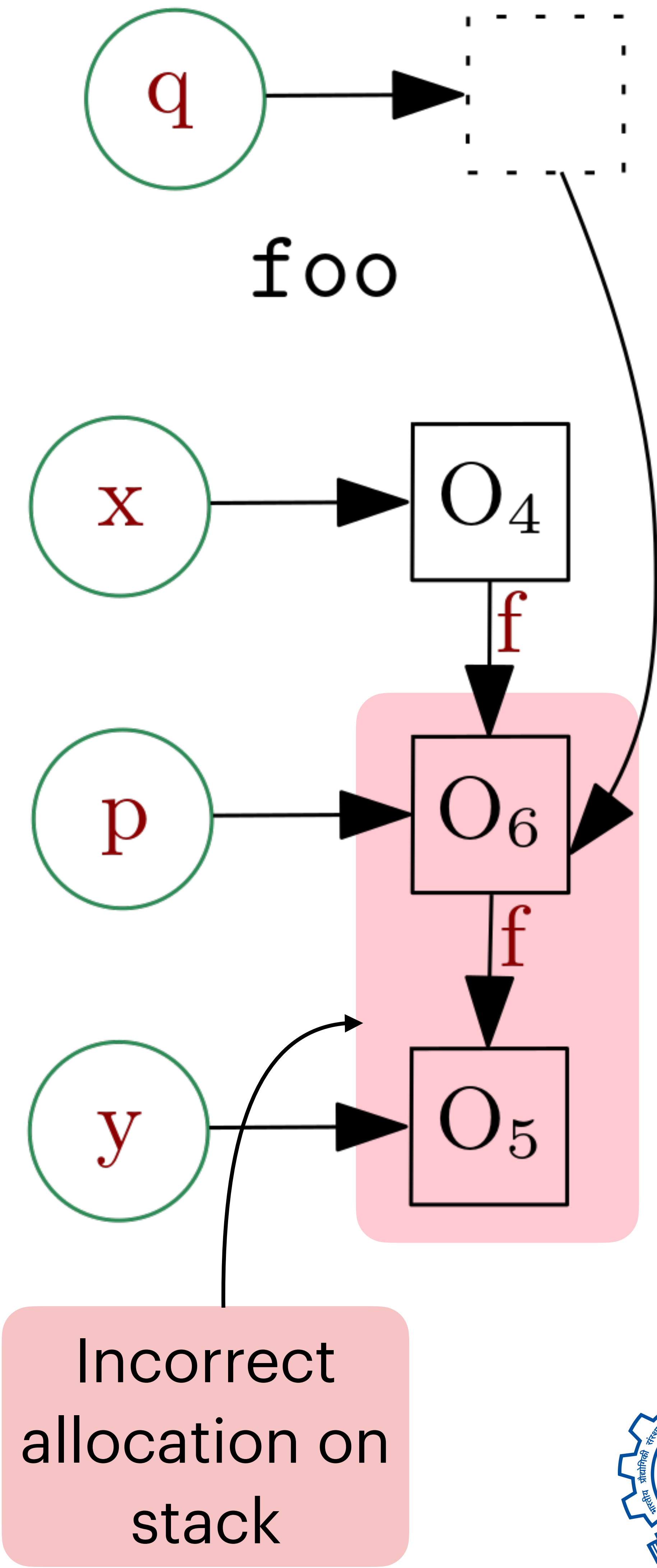


HCR Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
     q.f = p;
  }
} /* class A */
```

HCR Example

```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
  void zar(A p, A q) {
     q.f = p;
  }
} /* class A */
```



3. Callbacks

3. Callbacks

- Callback is a way to invoke an application method from a library method.

3. Callbacks

- Callback is a way to invoke an application method from a library method.
- Interprocedural static analysis requires a complete and precise callgraph.

3. Callbacks

- Callback is a way to invoke an application method from a library method.
- Interprocedural static analysis requires a complete and precise callgraph.
- Applications use large numbers of third party libraries.

3. Callbacks

- Callback is a way to invoke an application method from a library method.
- Interprocedural static analysis requires a complete and precise callgraph.
- Applications use large numbers of third party libraries.
  - Statically we don't know if there exits a callback edge from those library to application.

3. Callbacks

- Callback is a way to invoke an application method from a library method.
- Interprocedural static analysis requires a complete and precise callgraph.
- Applications use large numbers of third party libraries.
  - Statically we don’t know if there exits a callback edge from those library to application.
  - Callgraph remains imprecise statically.

3. Callbacks

- Callback is a way to invoke an application method from a library method.
- Interprocedural static analysis requires a complete and precise callgraph.
- Applications use large numbers of third party libraries.
  - Statically we don't know if there exits a callback edge from those library to application.
  - Callgraph remains imprecise statically.
- Using static analysis results during JIT compilation in presence of callbacks may give unsound results.

Callback Example

App1

```
public void foo1 (A p1) {
      . . . .
      A x = new A();
      this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
      . . . .
      A x = new A();
      this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
      . . . .
      A x = new A();
      this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
      . . . .
      A x = new A();
      this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```

Callback Example

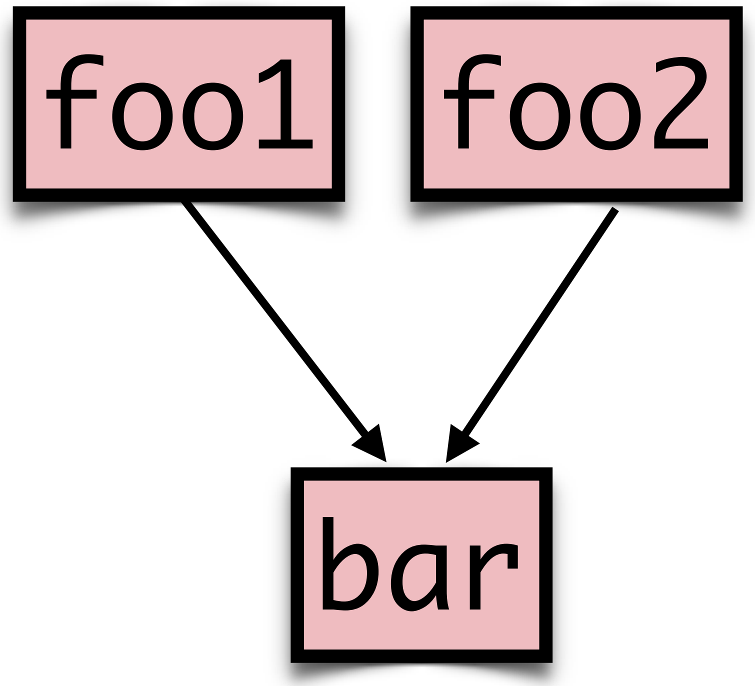
App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



Callback Example

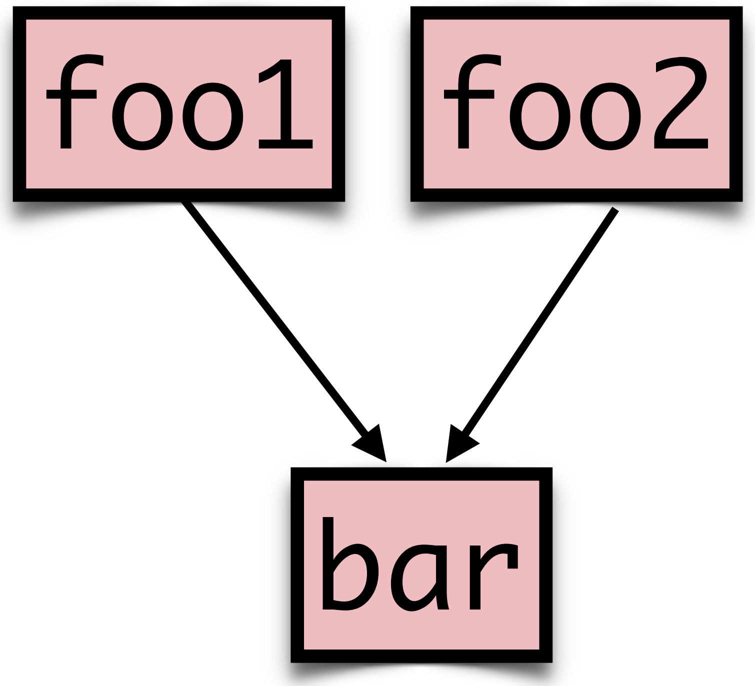
App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



O_4 can be stack allocated if bar gets inlined

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

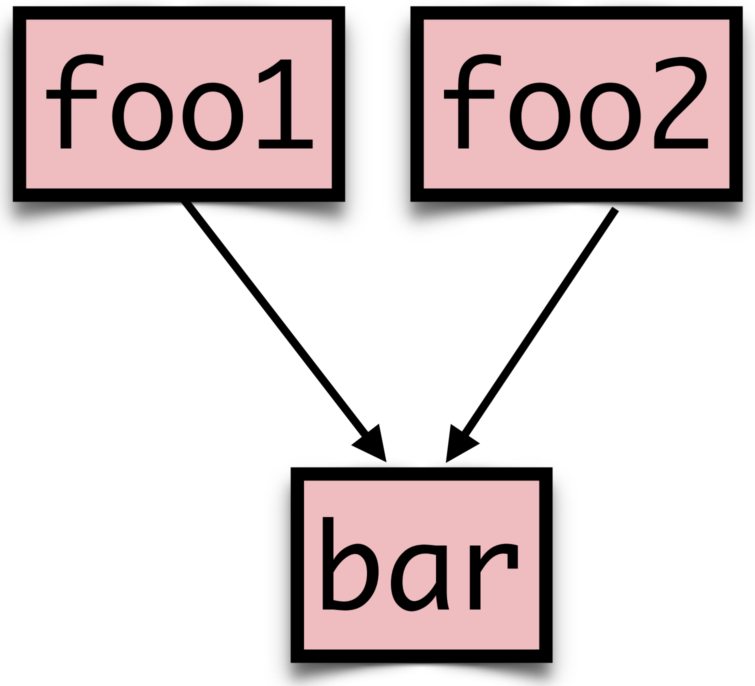
App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

Lib

```
public void lib () {
    A x = new A();
    global = x; //Escapes
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



O_4 can be stack allocated if bar gets inlined

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

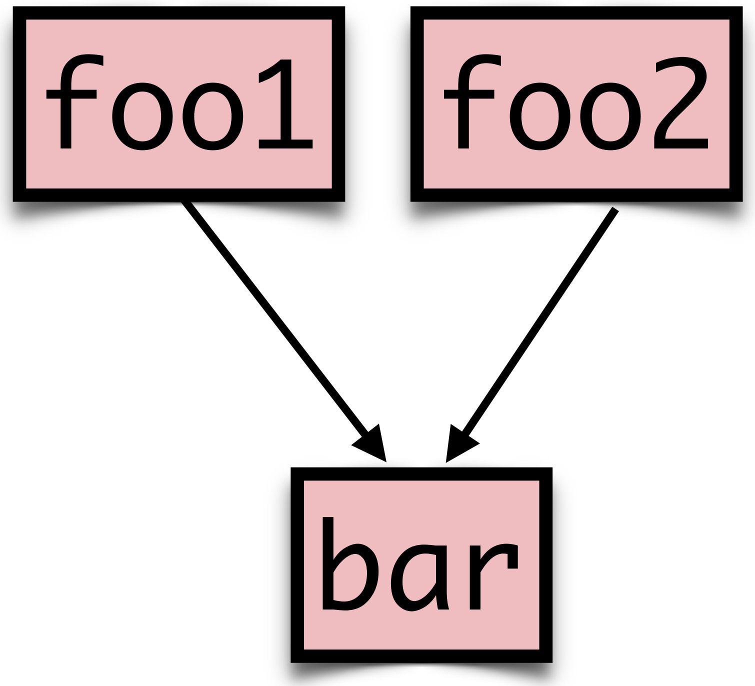
App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

Lib

```
public void lib () {
    A x = new A();
    global = x; //Escapes
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



O_4 can be stack allocated if bar gets inlined

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

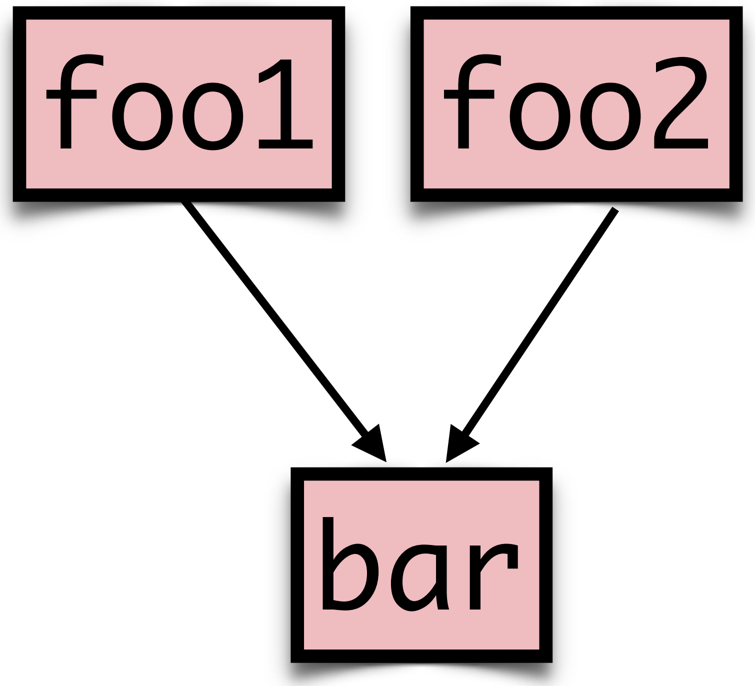
App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

Lib

```
public void lib () {
    A x = new A();
    global = x; //Escapes
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



O_4 can be stack allocated if bar gets inlined

Callback Example

App1

```
public void foo1 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

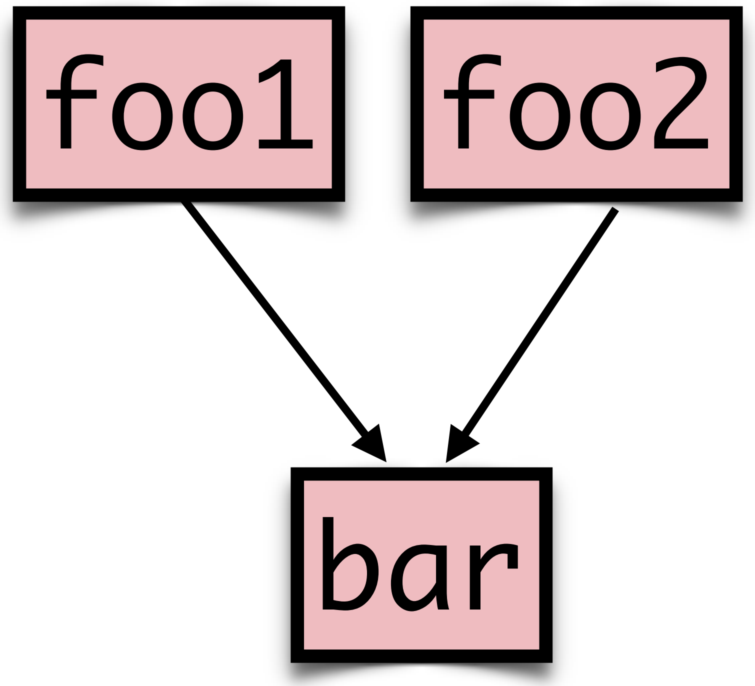
App2

```
public void foo2 (A p1) {
    . . . .
    A x = new A();
    this.bar(x)
}
```

Lib

```
public void lib () {
    A x = new A();
    global = x; //Escapes
    this.bar(x)
}
```

```
class A extends Library {
    @Override
    void bar(T p1) {
        A a = new A(); // O4
        p1.f = a;
    }
}
```



Incorrect stack allocation of O_4 in Lib.

4. Tampered Static-Analysis Results

4. Tampered Static-Analysis Results

- Static Analysis results — Unsound.

4. Tampered Static-Analysis Results

- Static Analysis results — Unsound.
  - Missed corner cases.

4. Tampered Static-Analysis Results

- Static Analysis results — Unsound.
  - Missed corner cases.
  - Altered result files.

4. Tampered Static-Analysis Results

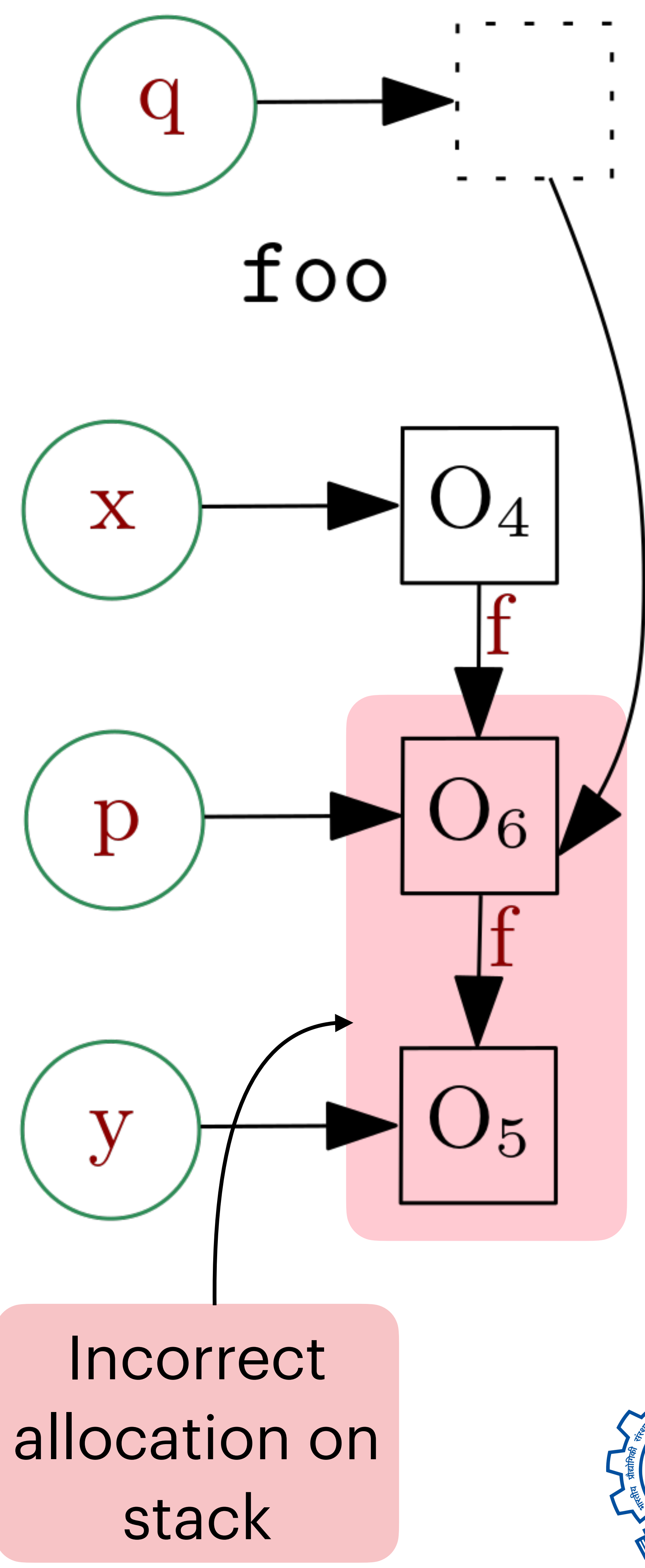
- Static Analysis results — Unsound.
  - Missed corner cases.
  - Altered result files.

- **How to safely allocate objects on stack in a managed runtime using static-analysis results?**

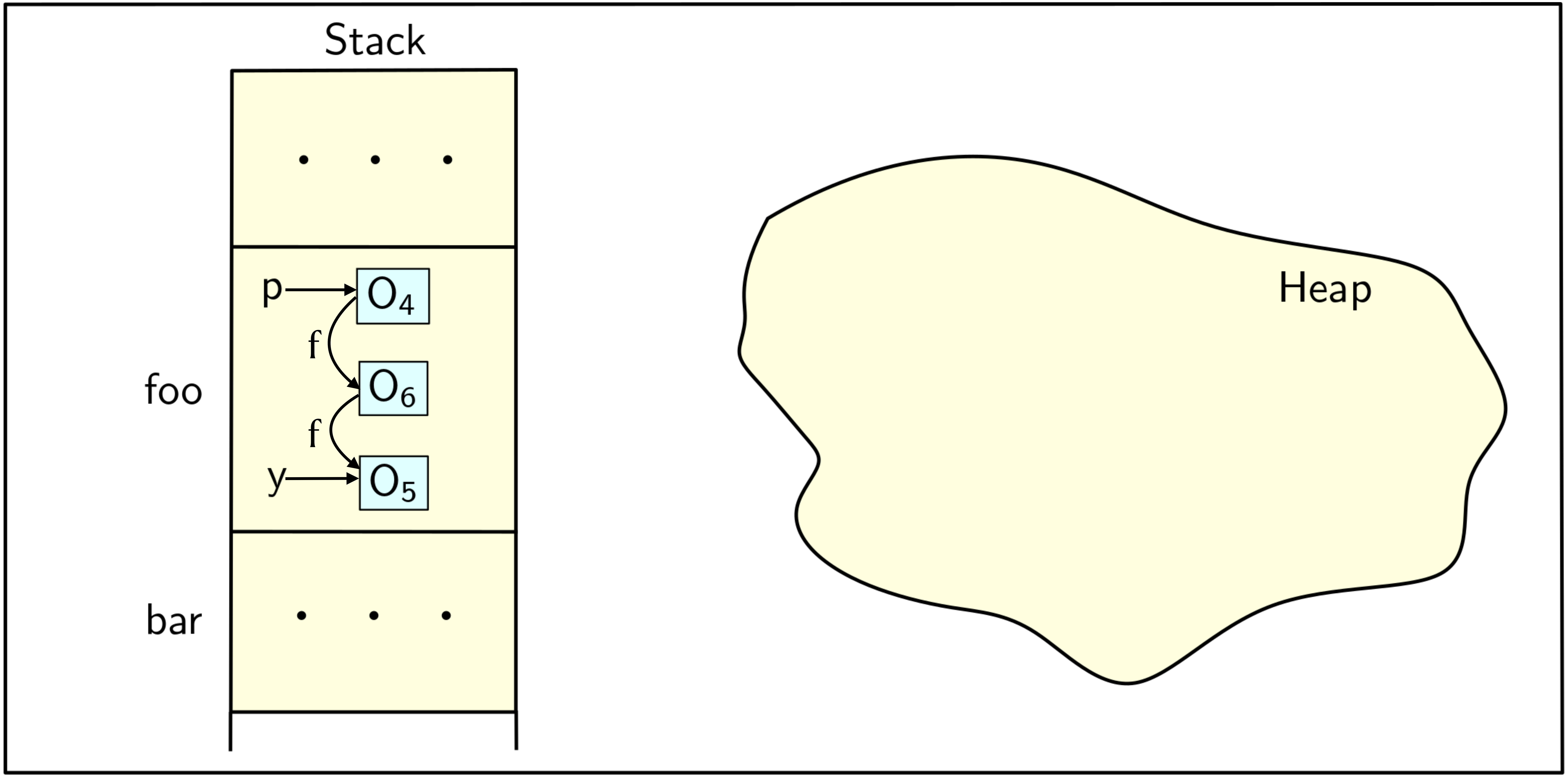
Dynamic Heapification

Dynamic Class Loading Example

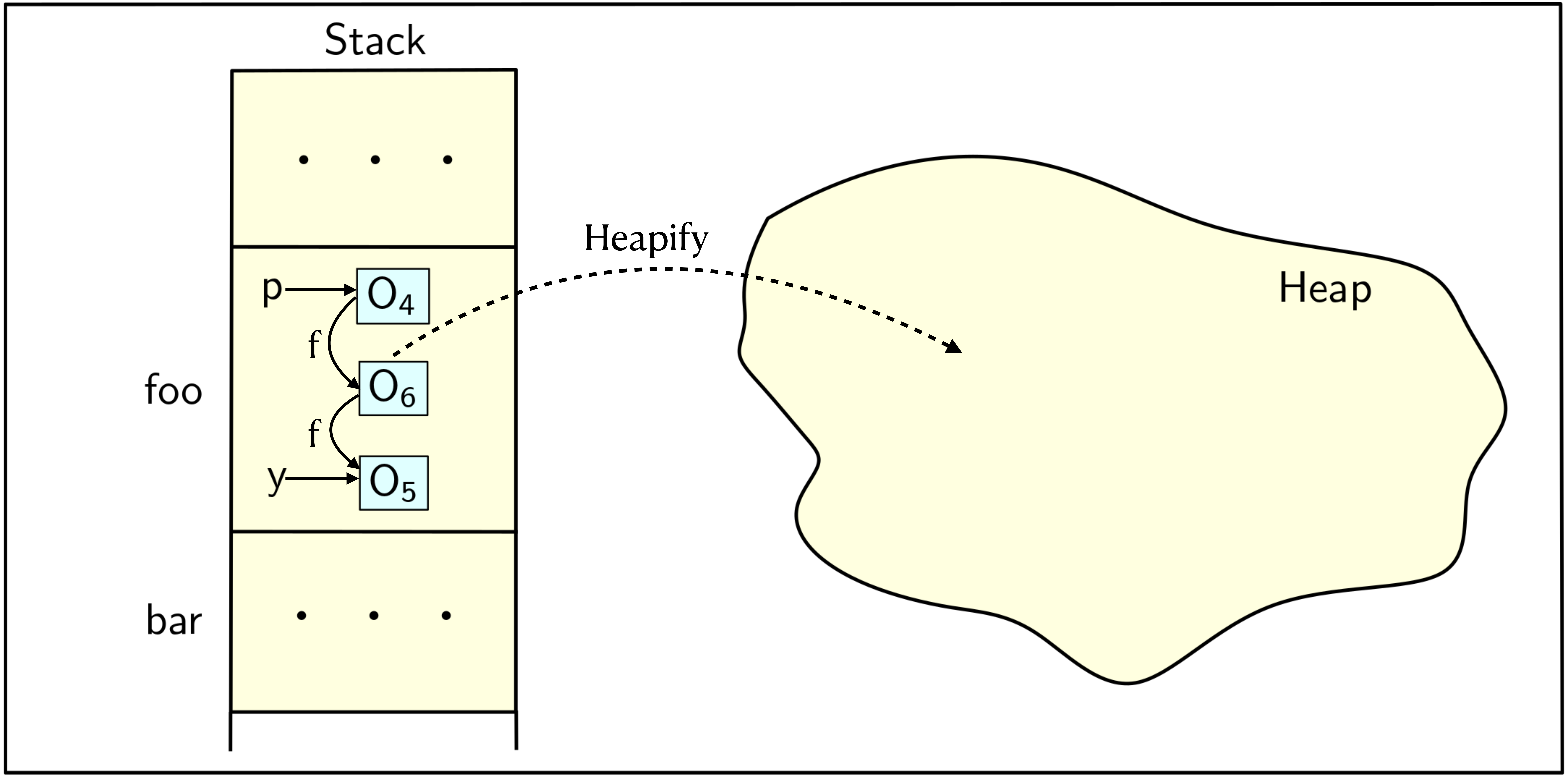
```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A */
class B extends A
  void zar(A p, A q) {
     q.f = p;
  } /* method zar */
} /* class B */
```



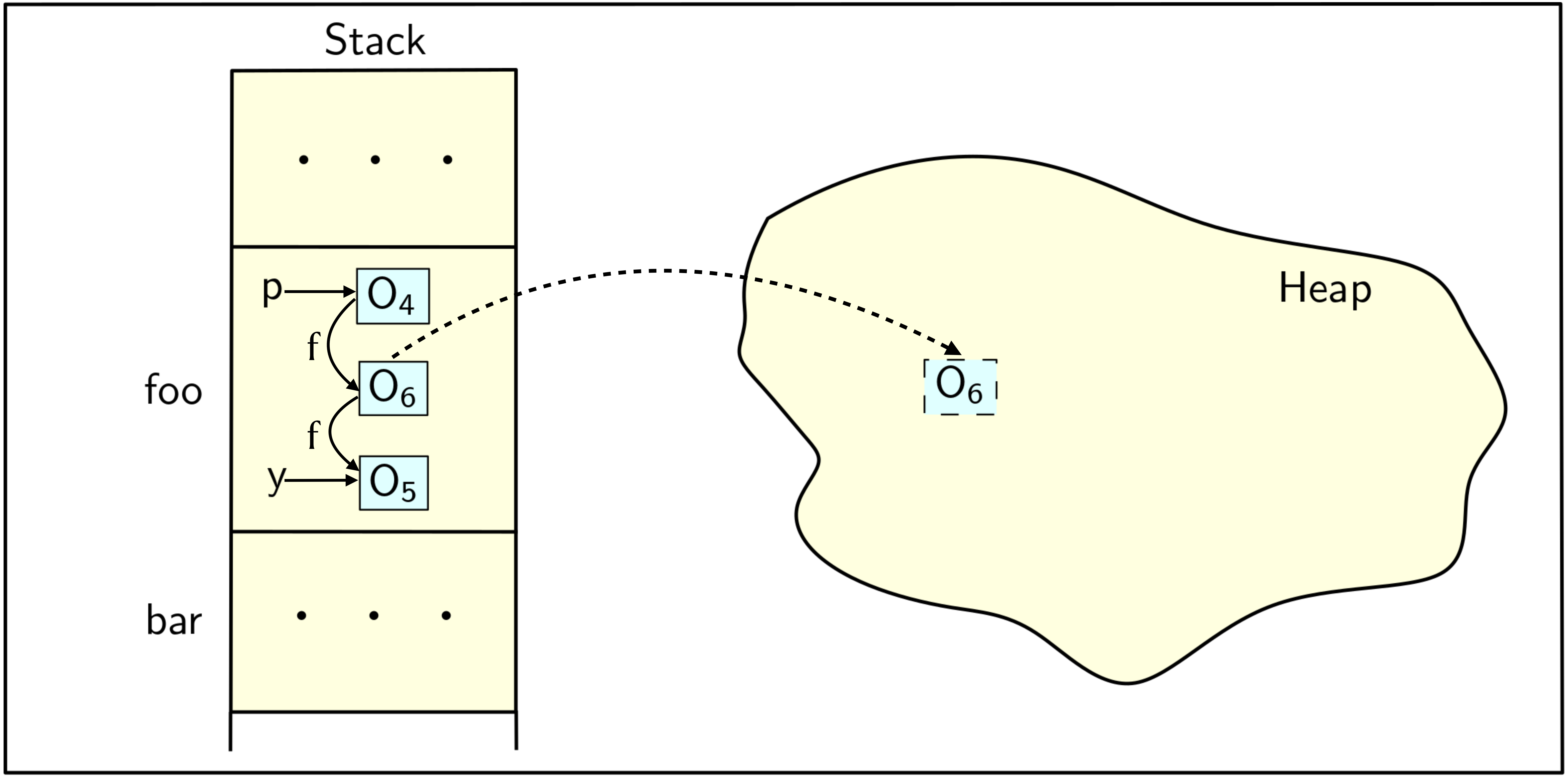
Heapification



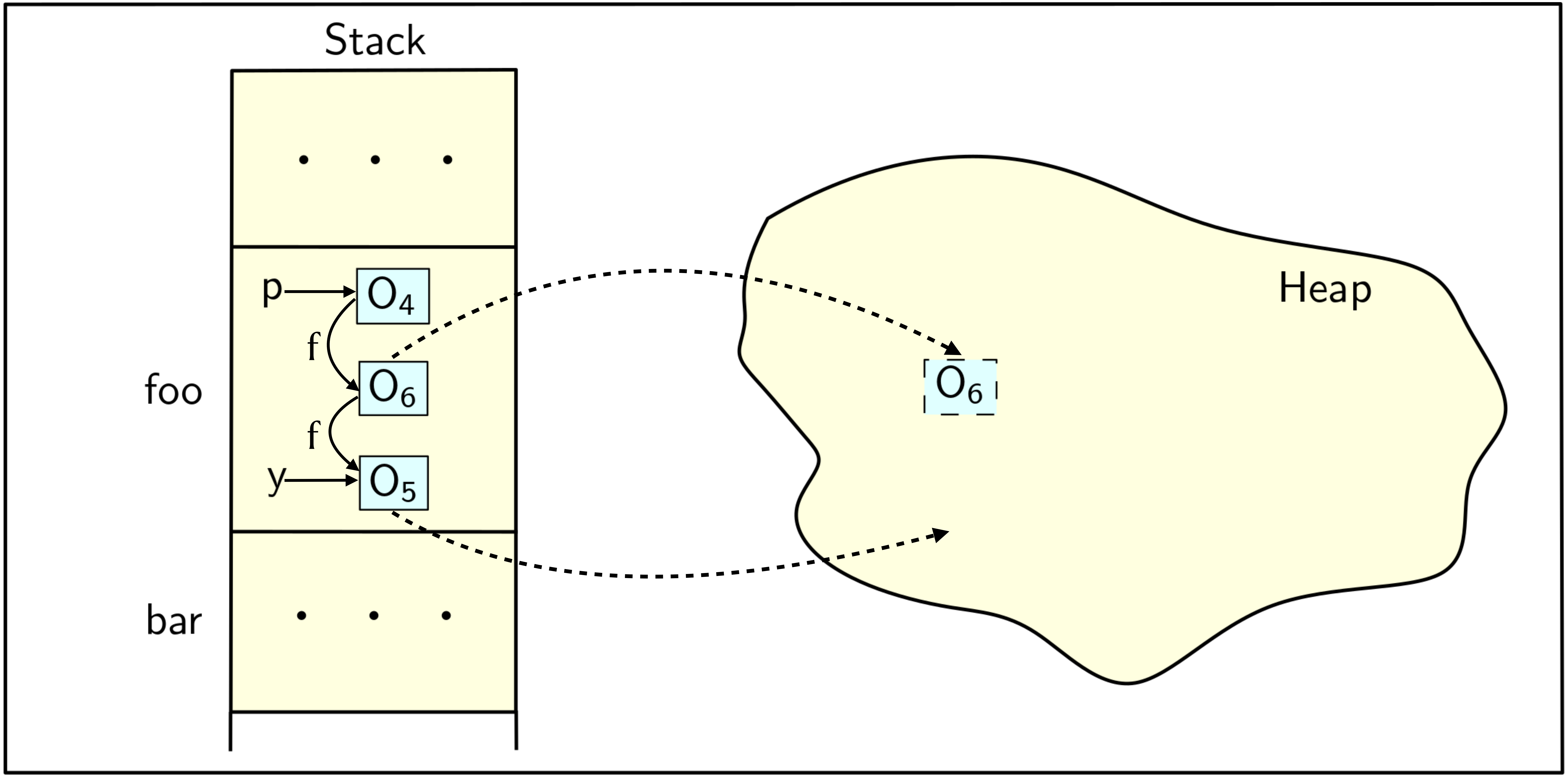
Heapification



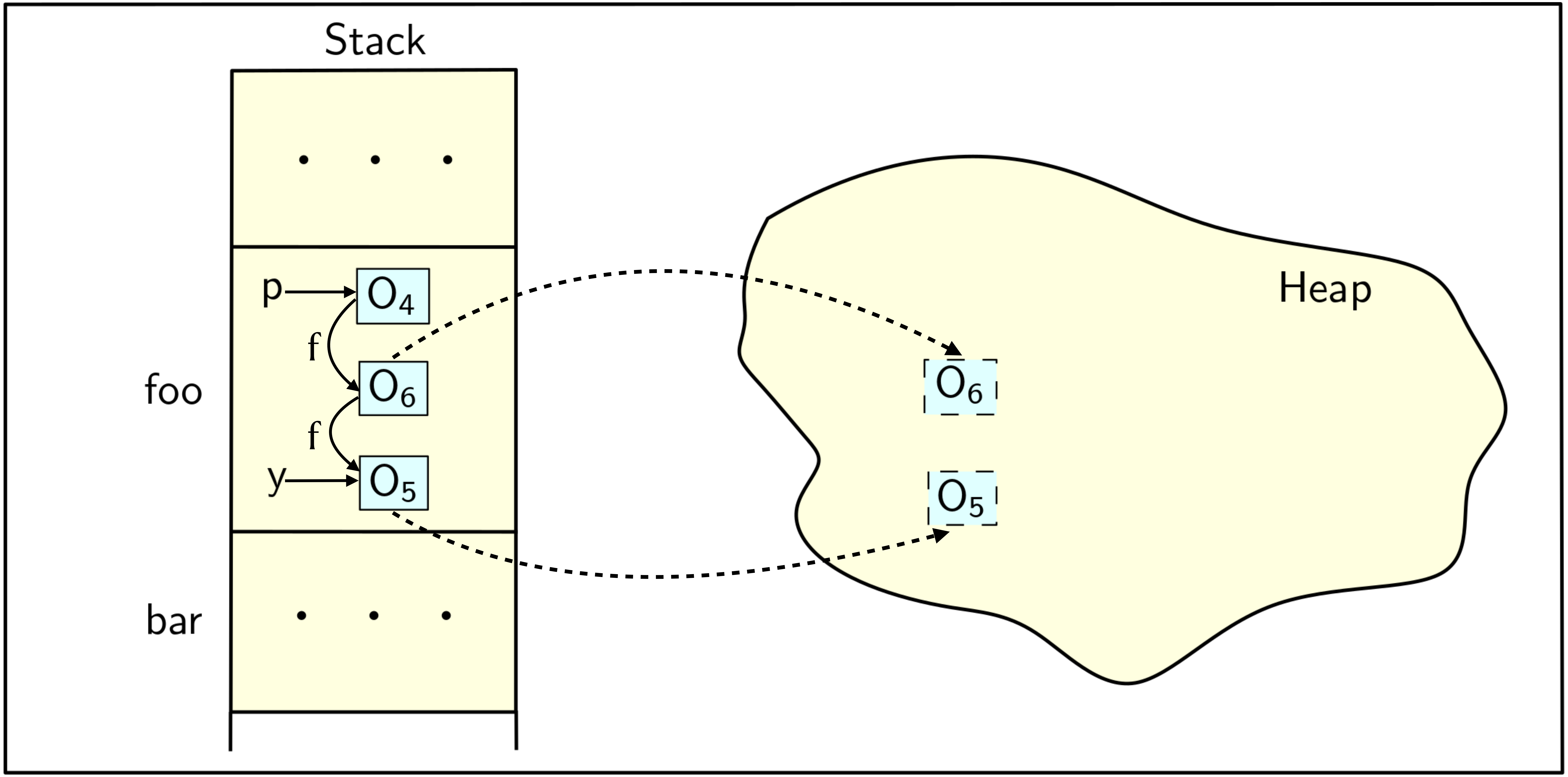
Heapification



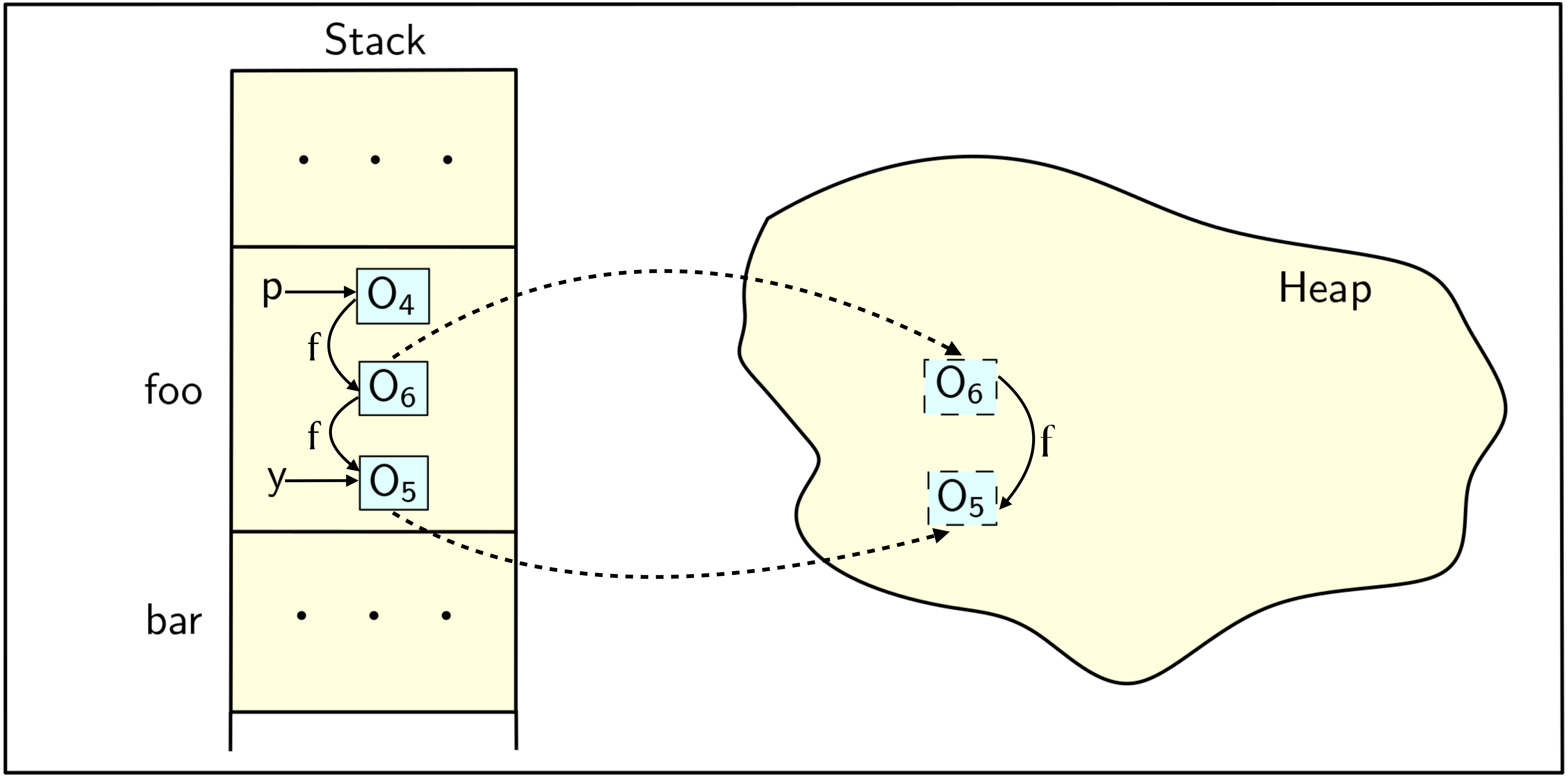
Heapification



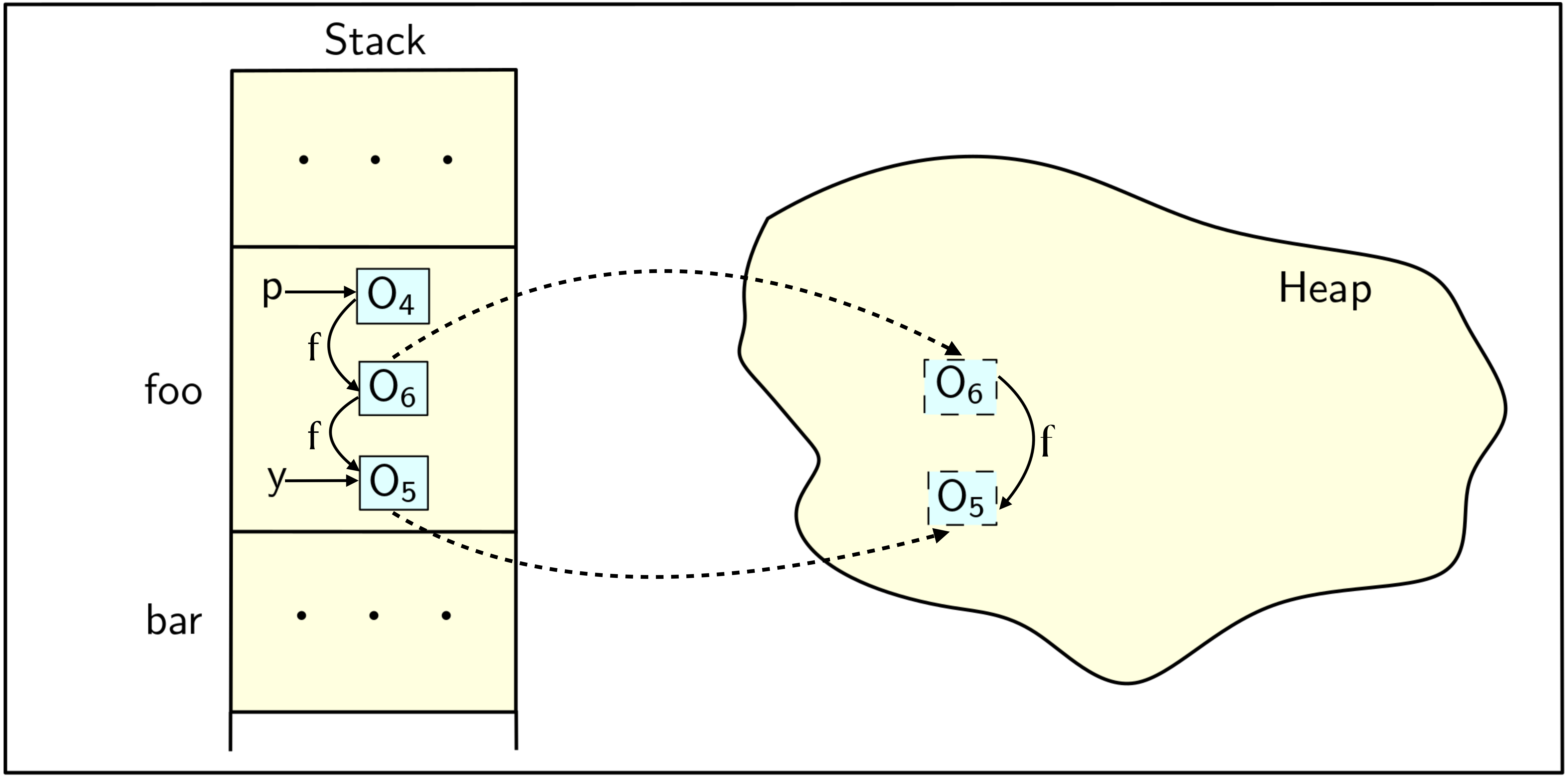
Heapification



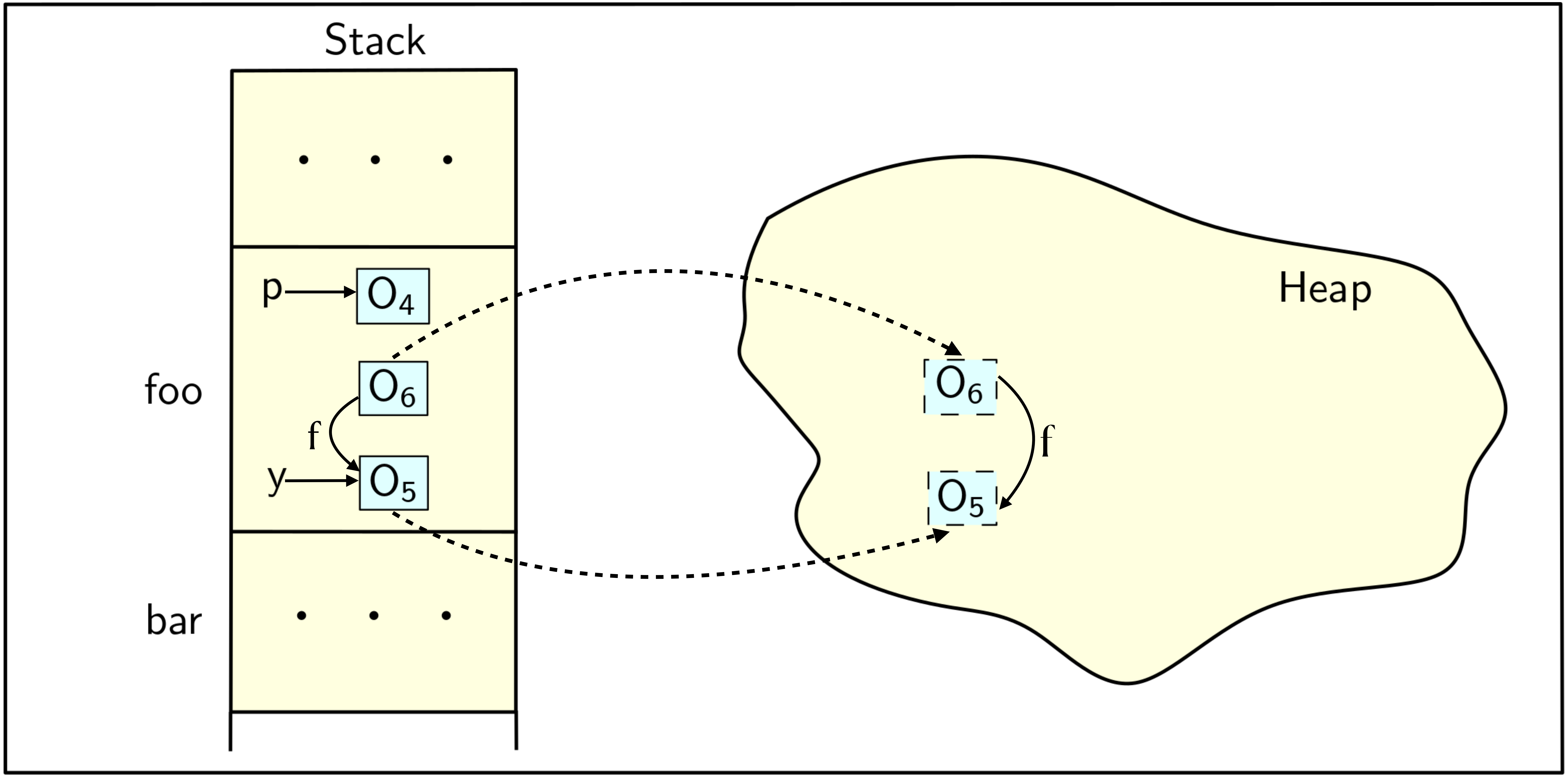
Heapification



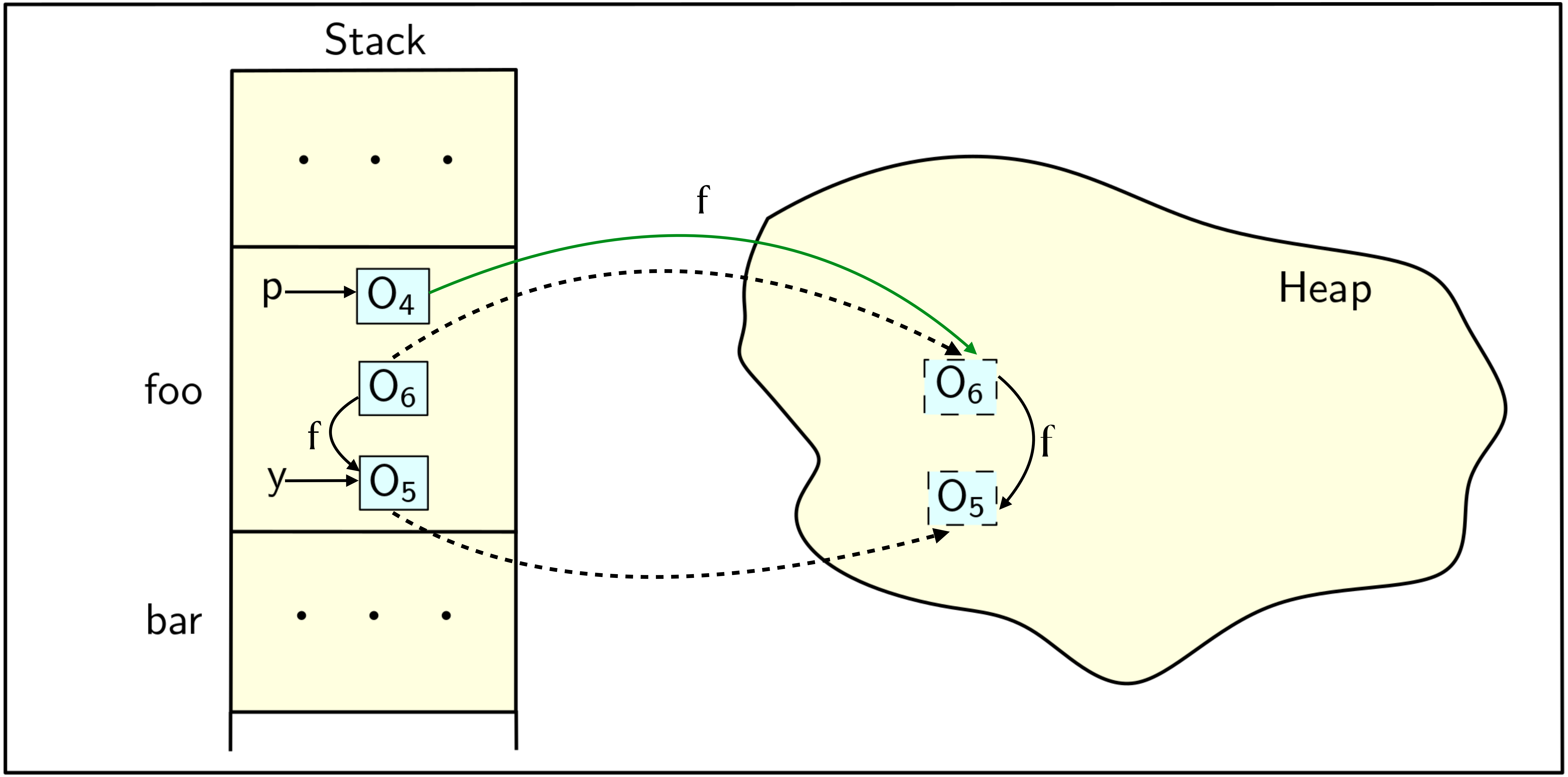
Heapification



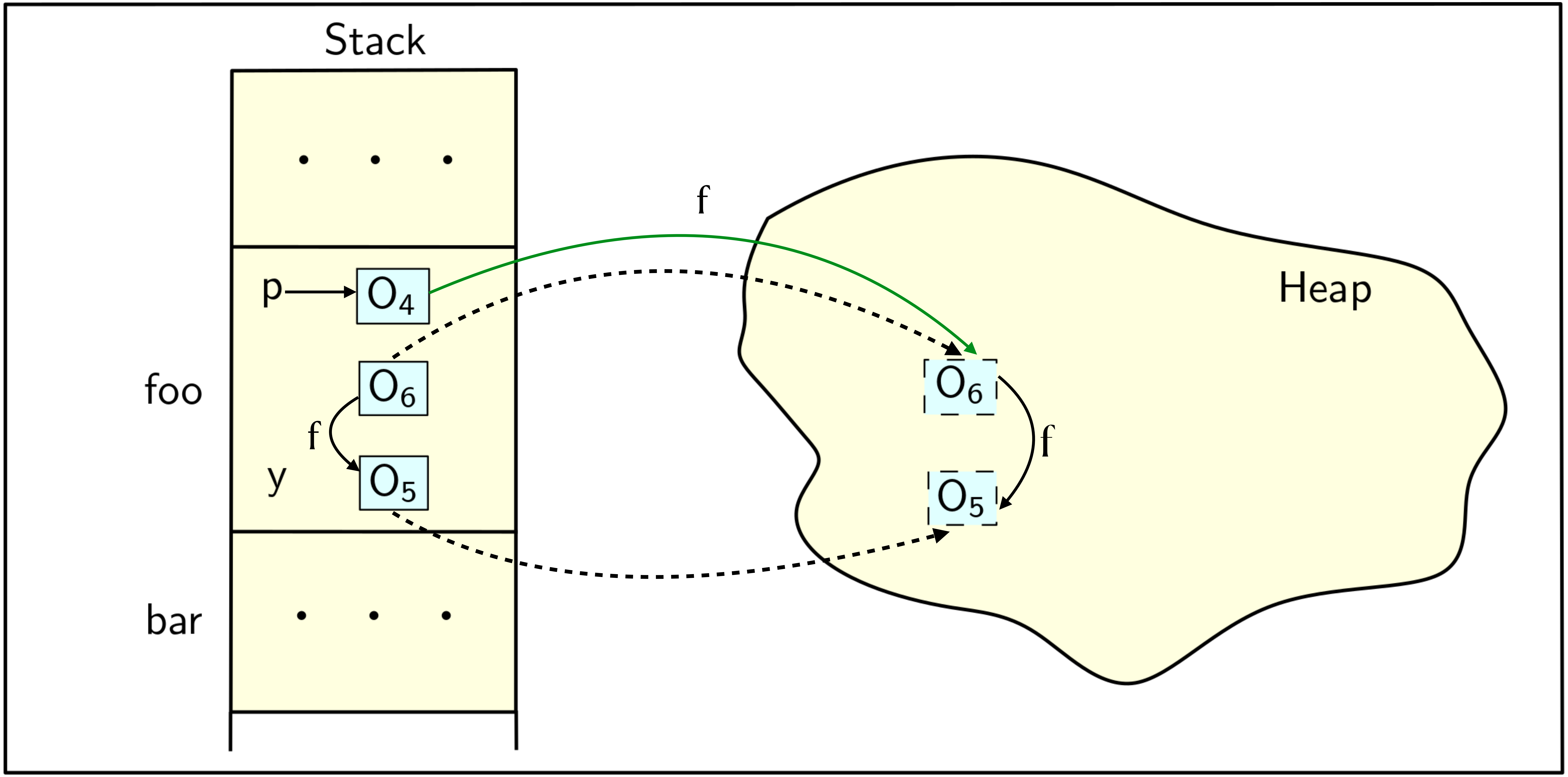
Heapification



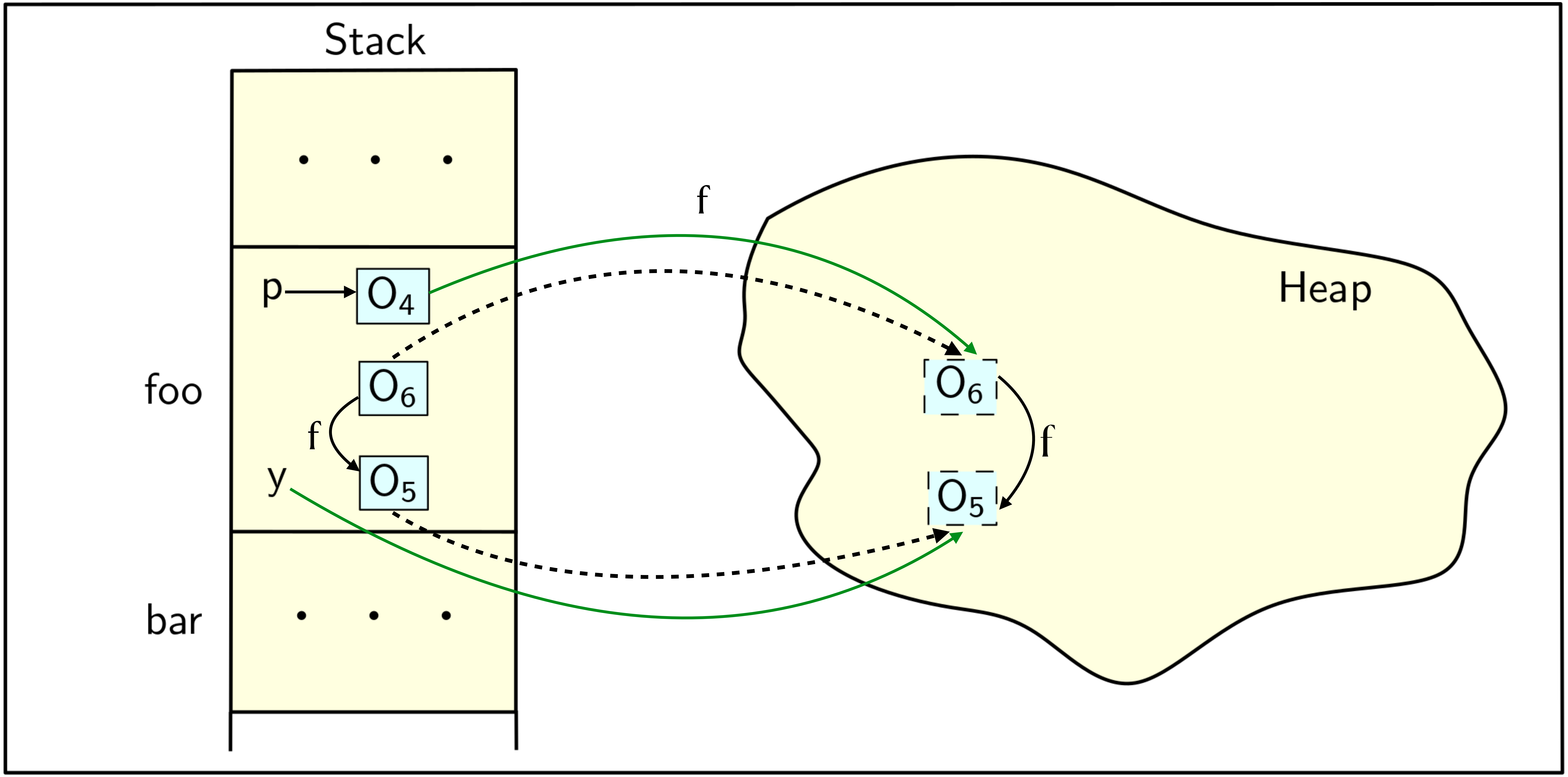
Heapification



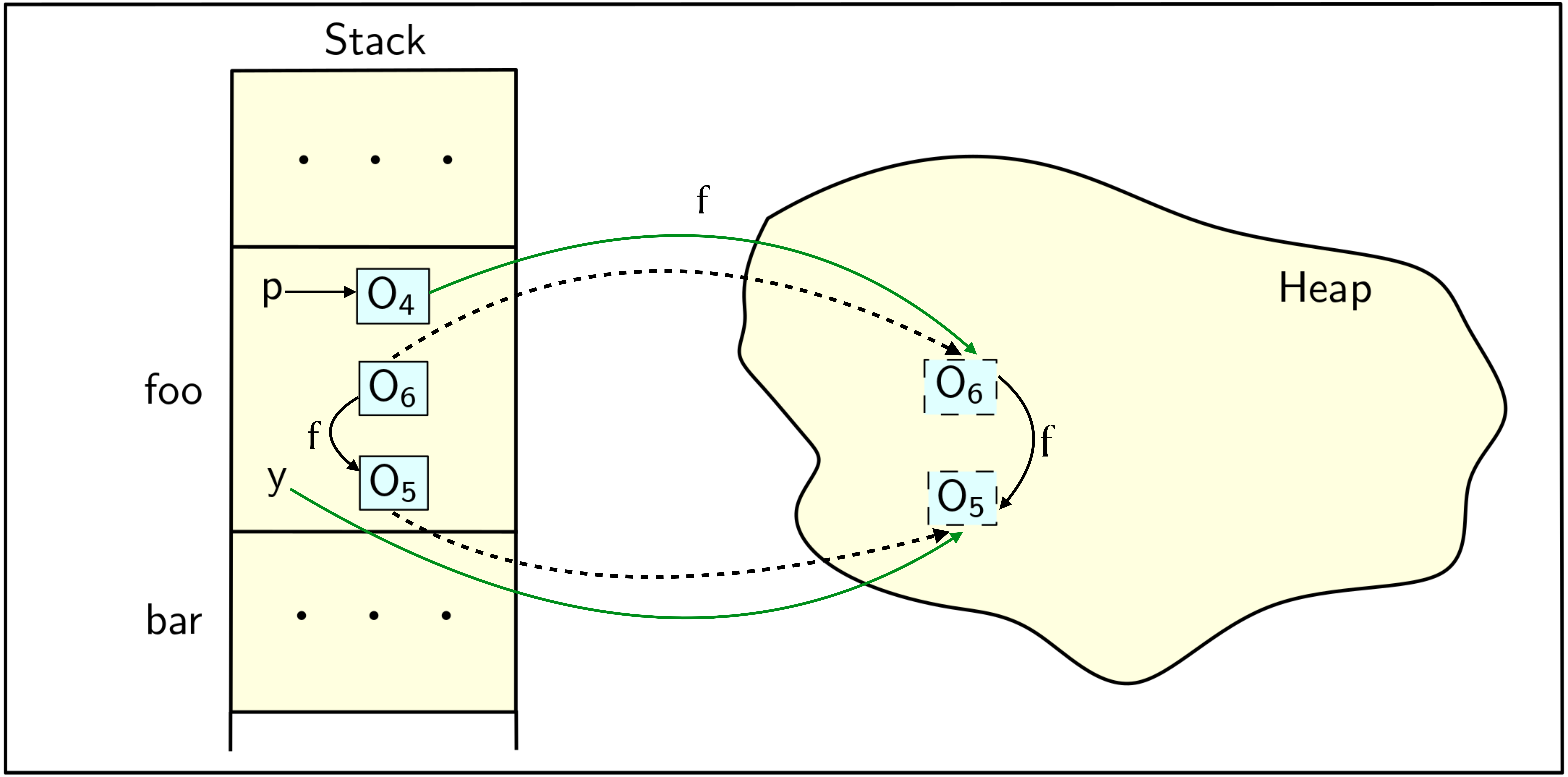
Heapification



Heapification



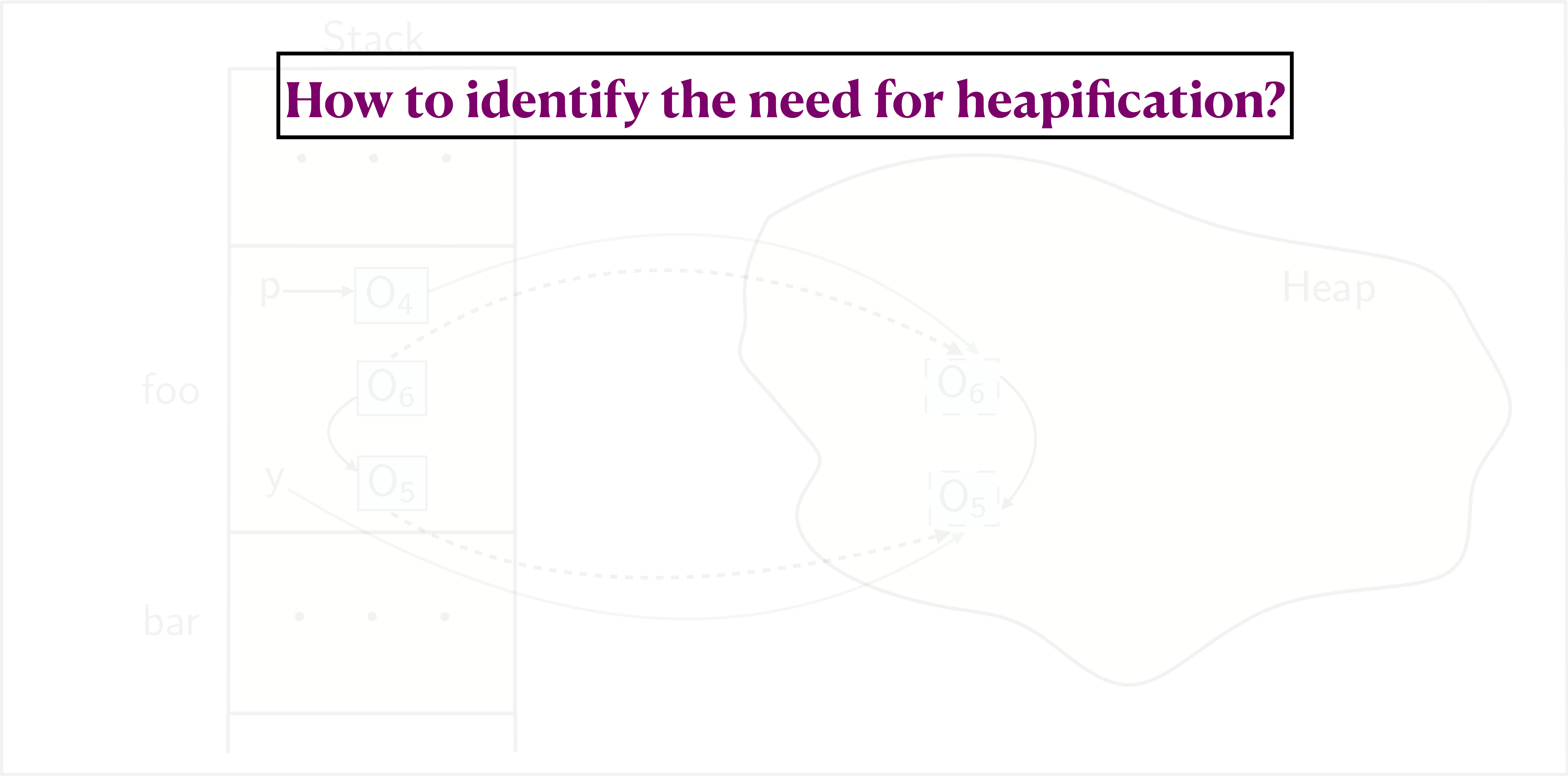
Heapification



Heapification

Heapification

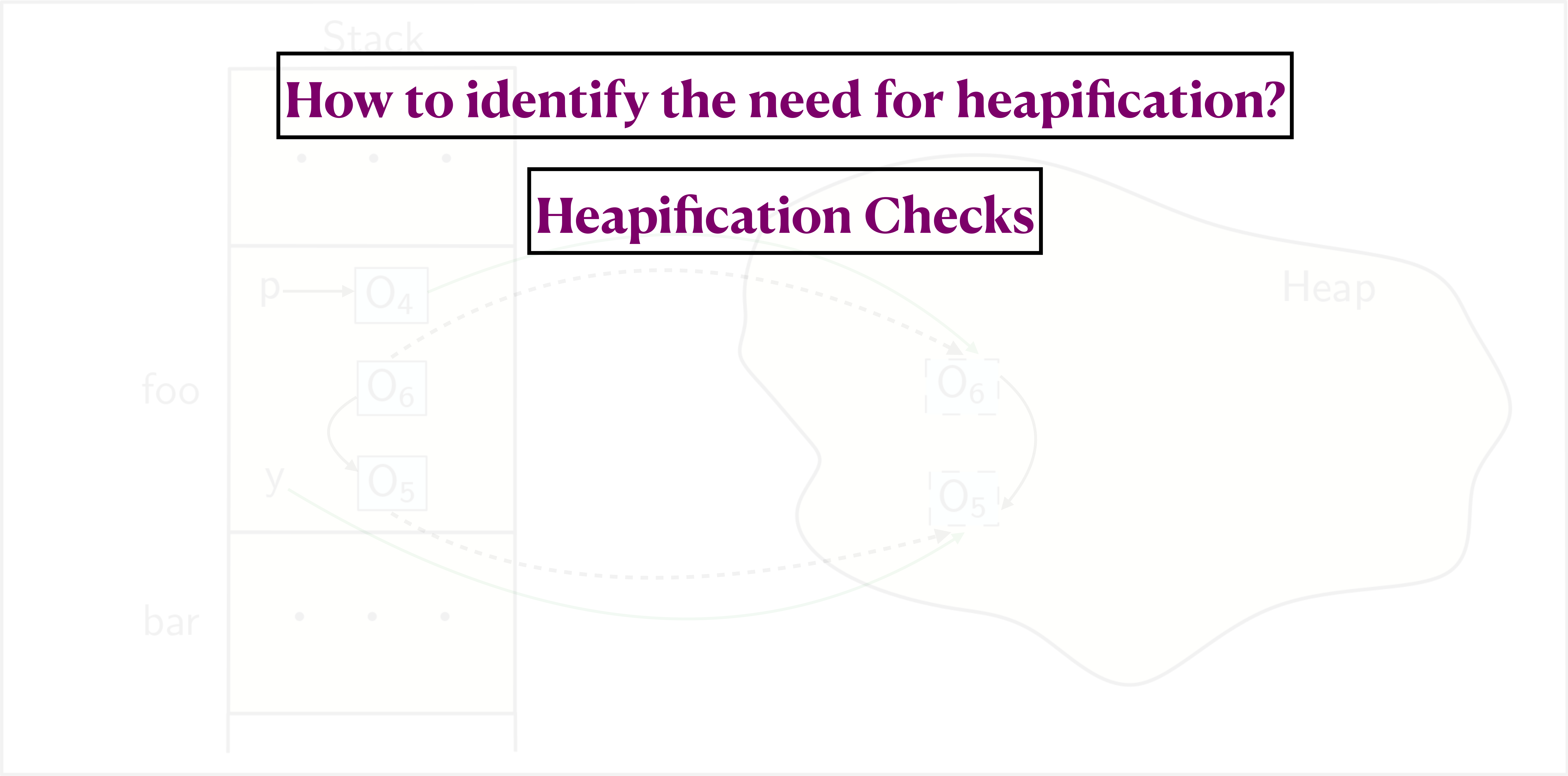
How to identify the need for heapification?



Heapification

How to identify the need for heapification?

Heapification Checks



Heapification

How to identify the need for heapification?

Heapification Checks

- Return of references. (Byte code: `return`.)
- References stores. (Byte code: `putfield, putstatic, aastore`.)
- Throwing of exception. (Byte code: athrow.)
- Calls to native. (Byte code: putObject, putObjectOrdered, putObjectVolatile.)
- JNI APIs used to perform stores in called C/C++ code. (Byte code: setObjectField.)

Checking the Need for Heapification

```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if lhs object has longer life time than rhs object then
                Heapify the rhs object.
```

Checking the Need for Heapification

```
a.f = b;
```

lhs_obj rhs_obj

Procedure HeapificationCheckAtStore(`lhs`, `rhs`)
- **if** *rhs object is outside stack bounds* **then**
  - No heapification required.
- **else**
  - /* The rhs object is present on the stack */
  - **if** *lhs object is outside stack bounds* **then**
    - Heapify starting from the rhs object.
  - **else**
    - /* Both lhs and rhs objects are on the stack */
    - **if** *lhs object has longer life time than rhs object* **then**
      - Heapify the rhs object.

Checking the Need for Heapification

```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if lhs object has longer life time than rhs object then
                Heapify the rhs object.
```

```
a.f = b;
```

lhs_obj

rhs_obj

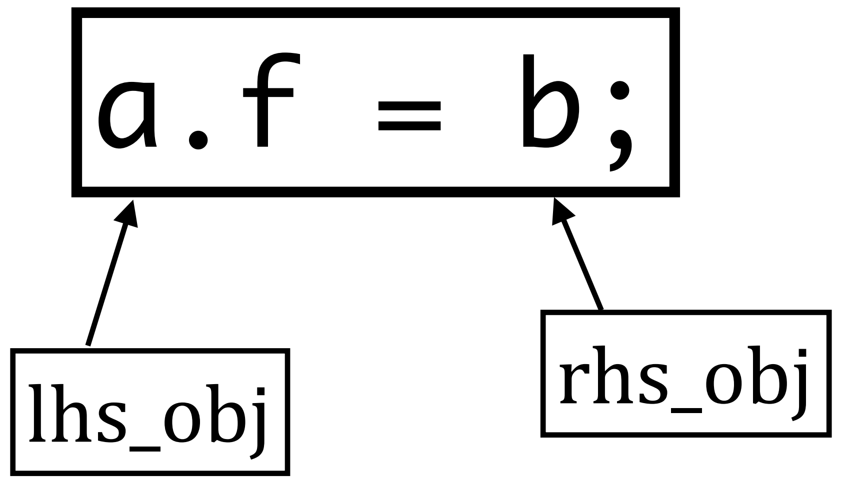
Checking the Need for Heapification

```
a.f = b;
```

lhs_obj rhs_obj

```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if lhs object has longer life time than rhs object then
                Heapify the rhs object.
```

Checking the Need for Heapification




```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if lhs object has longer life time than rhs object then
                Heapify the rhs object.
```

Checking the Need for Heapification

```
a.f = b;
```

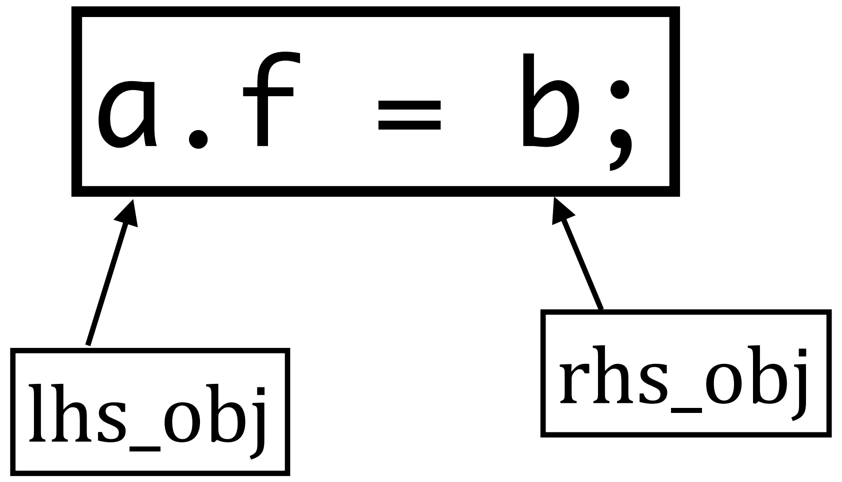
lhs_obj

rhs_obj

Procedure HeapificationCheckAtStore(`lhs`, `rhs`)

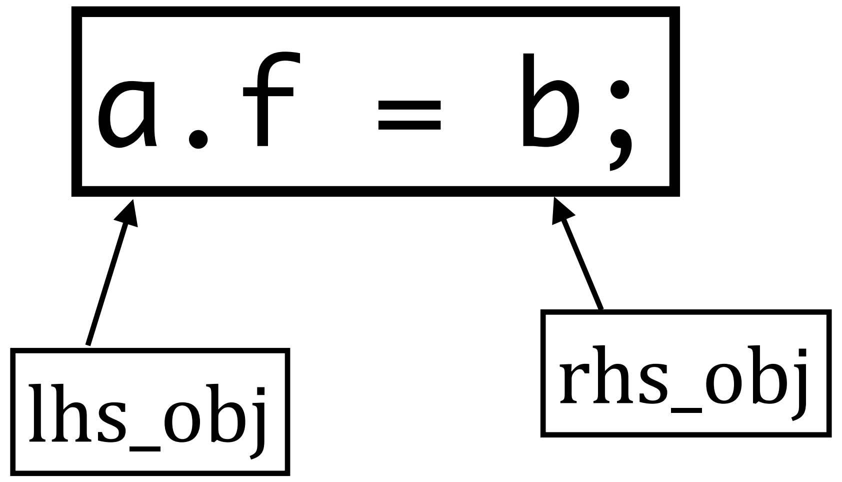
- **if** *rhs object is outside stack bounds* **then**
  - No heapification required.
- **else**
  - /* The rhs object is present on the stack */
  - **if** *lhs object is outside stack bounds* **then**
    - Heapify starting from the rhs object.
  - **else**
    - /* Both lhs and rhs objects are on the stack */
    - **if** *lhs object has longer life time than rhs object* **then**
      - Heapify the rhs object.

Checking the Need for Heapification



Procedure HeapificationCheckAtStore(`lhs`, `rhs`)
- **if** *rhs object is outside stack bounds* **then**
  - No heapification required.
- **else**
  - /* The rhs object is present on the stack */
  - **if** *lhs object is outside stack bounds* **then**
    - Heapify starting from the rhs object.
  - **else**
    - /* Both lhs and rhs objects are on the stack */
    - **if** *lhs object has longer life time than rhs object* **then**
      - Heapify the rhs object.

Checking the Need for Heapification



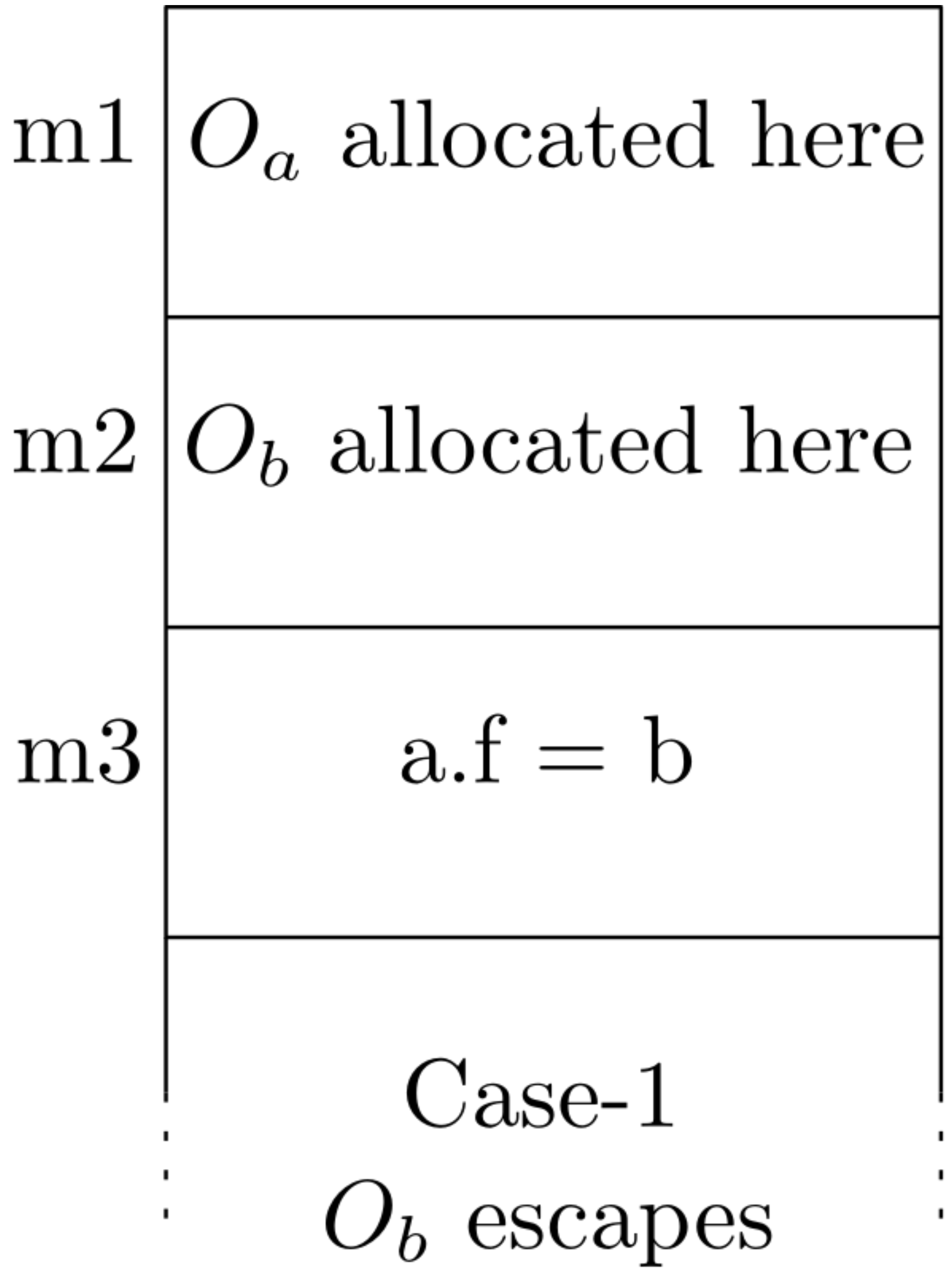

```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if lhs object has longer life time than rhs object then
                Heapify the rhs object.
```

Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
     a.f = b;
  } /* method m3 */
} /* class T */
```

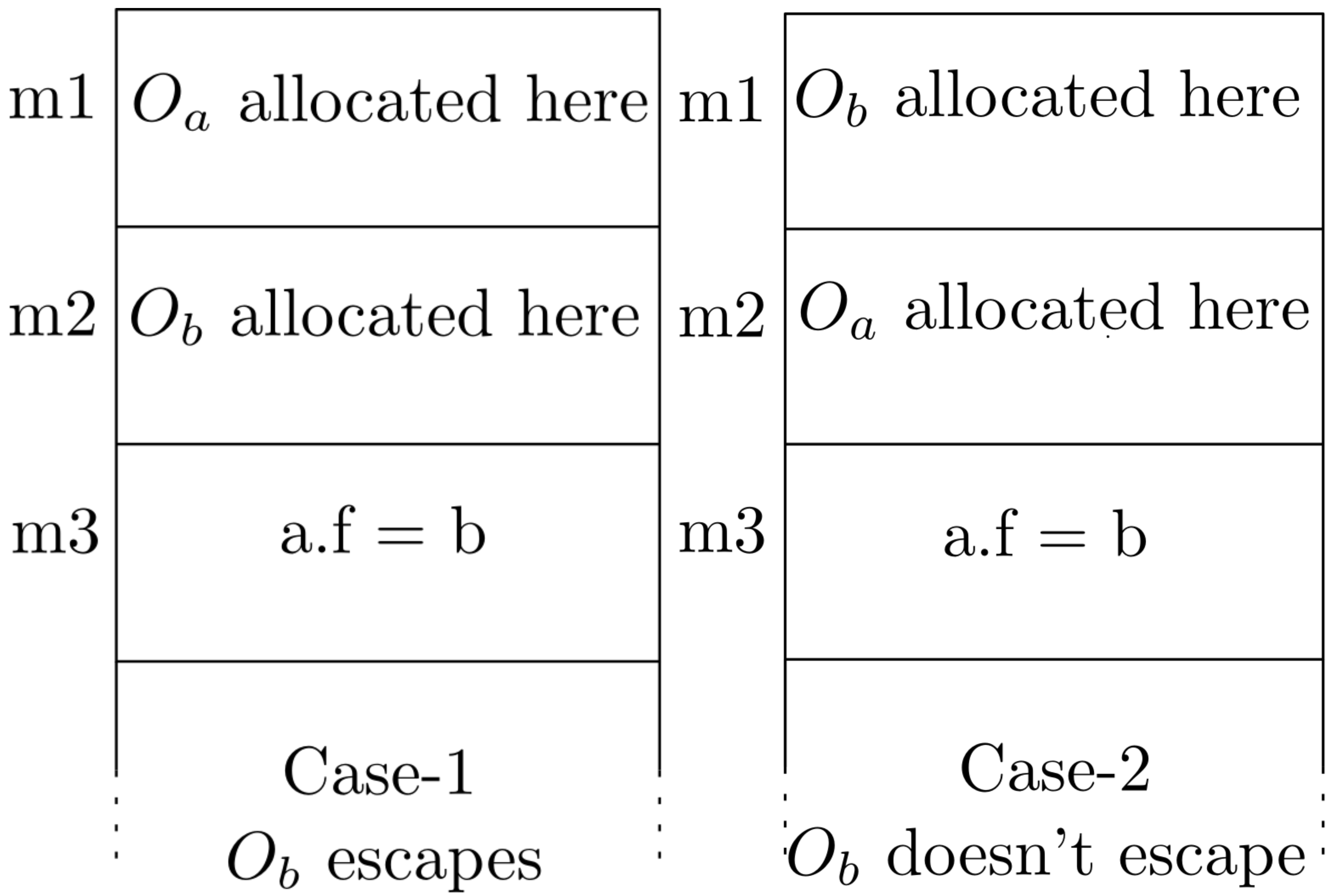
Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
      a.f = b;
  } /* method m3 */
} /* class T */
```



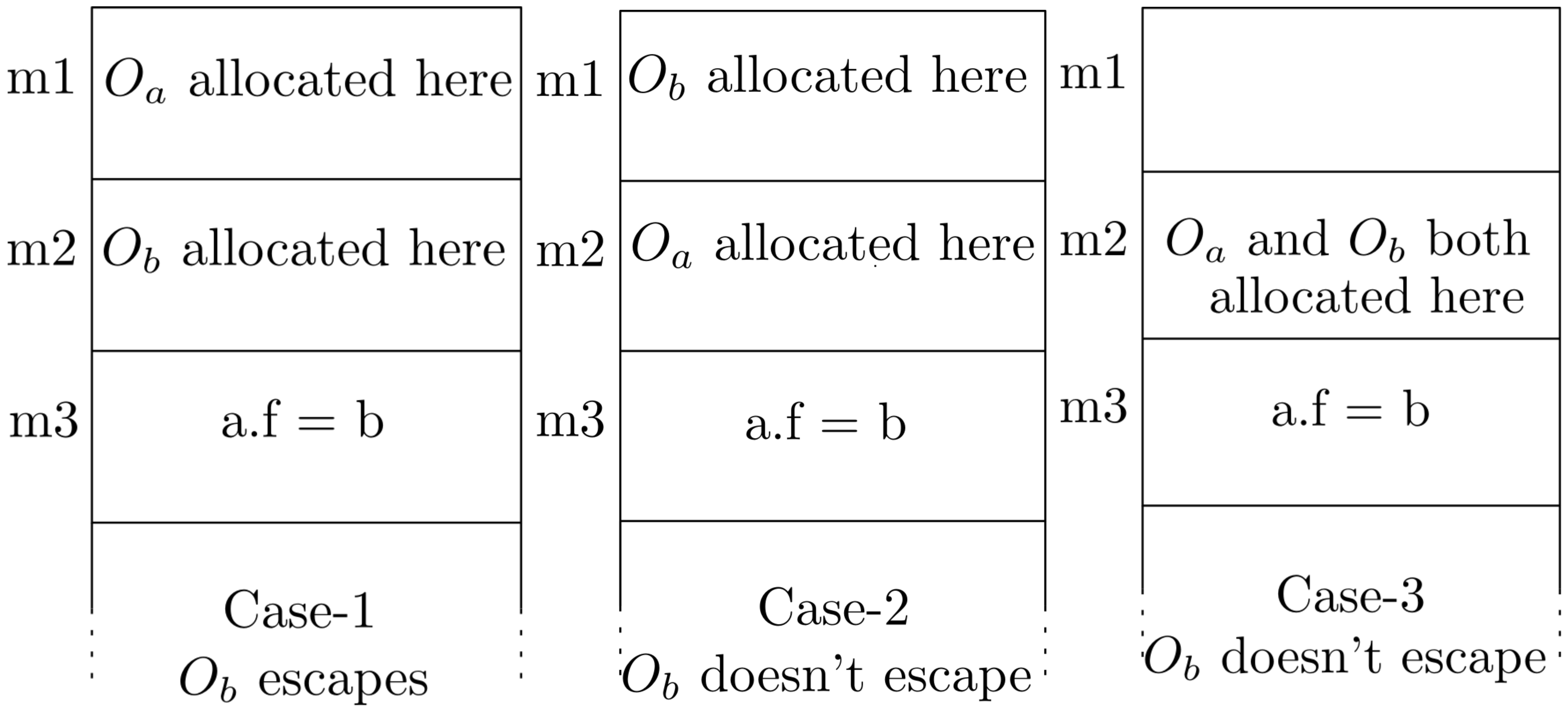
Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
      a.f = b;
  } /* method m3 */
} /* class T */
```



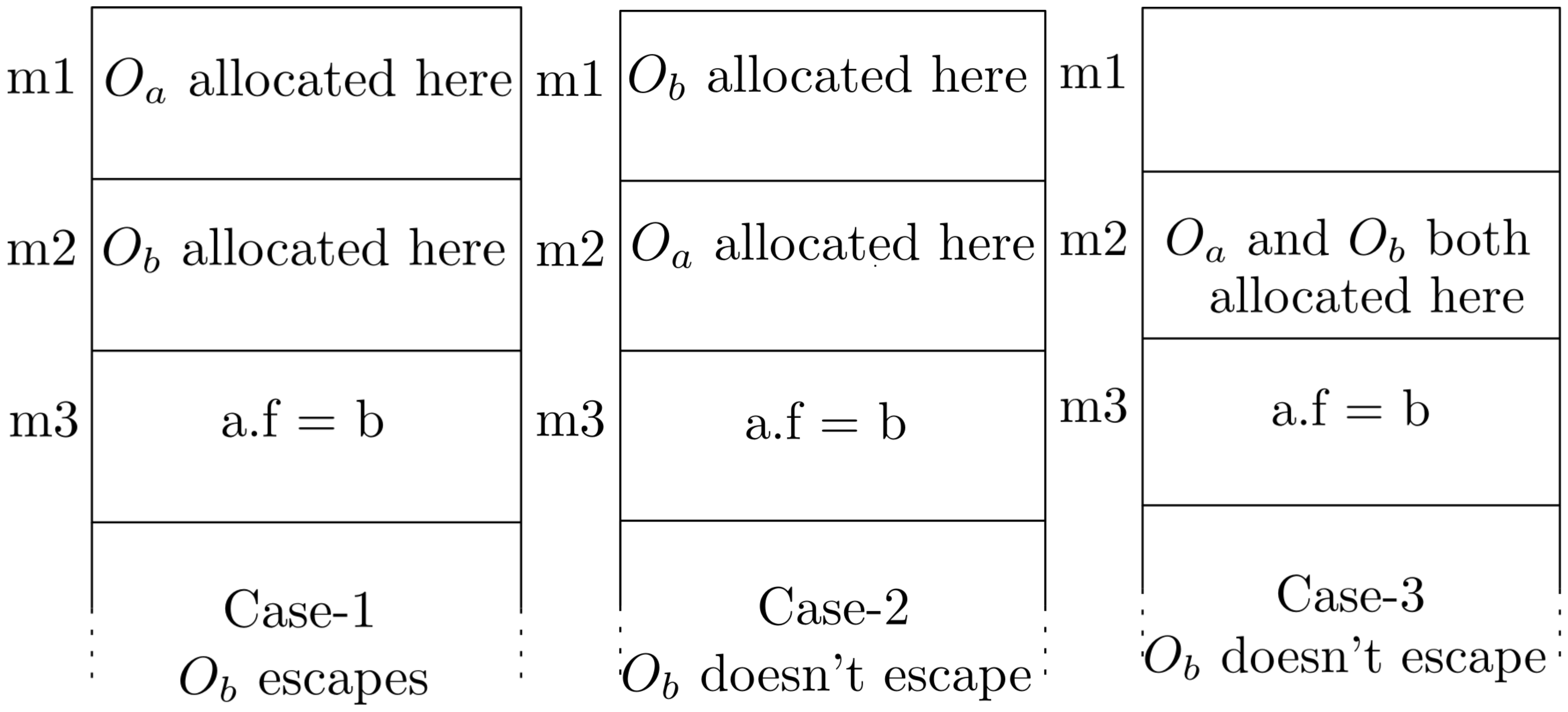
Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
      a.f = b;
  } /* method m3 */
} /* class T */
```



Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
     a.f = b;
  } /* method m3 */
} /* class T */
```



Stack Walk — Costly

Scenarios at Store Statement

```
class T {
  T f;
  void m1() {m2(. . .);}
  void m2() {m3(. . .);}
  void m3(T a, T b) {
     a.f = b;
  } /* method m3 */
} /* class T */
```

`rhs_obj >= lhs_obj`

	Case-1: O_b escapes	Case-2: O_b doesn't escape	Case-3: O_b doesn't escape
m1	O_a allocated here	O_b allocated here	
m2	O_b allocated here	O_a allocated here	O_a and O_b both allocated here
m3	a.f = b	a.f = b	a.f = b

Stack Walk — Costly

Ordering Objects on Stack

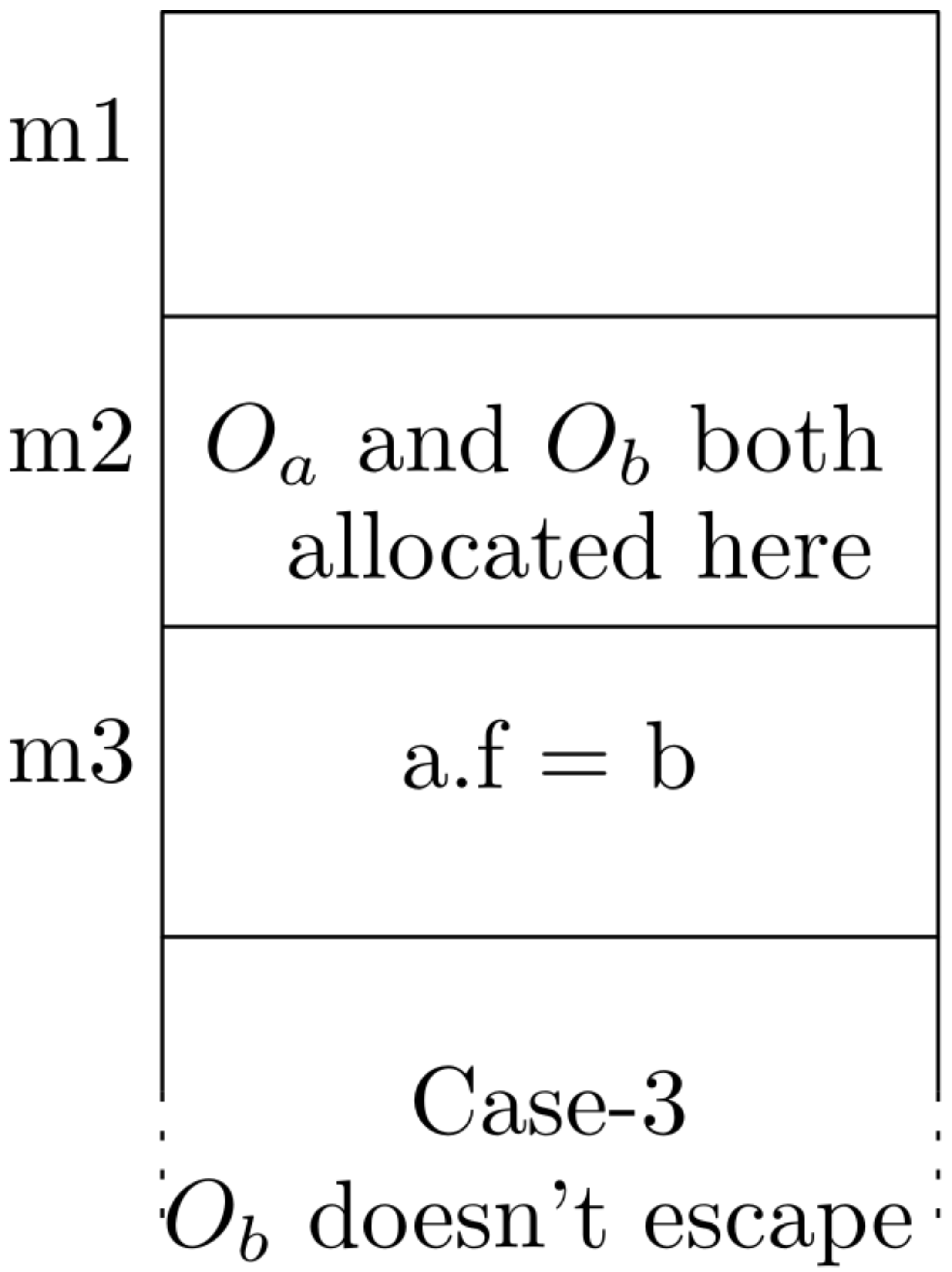
Ordering Objects on Stack

Ordering Objects on Stack

- A simple address-comparison check works majority of times.

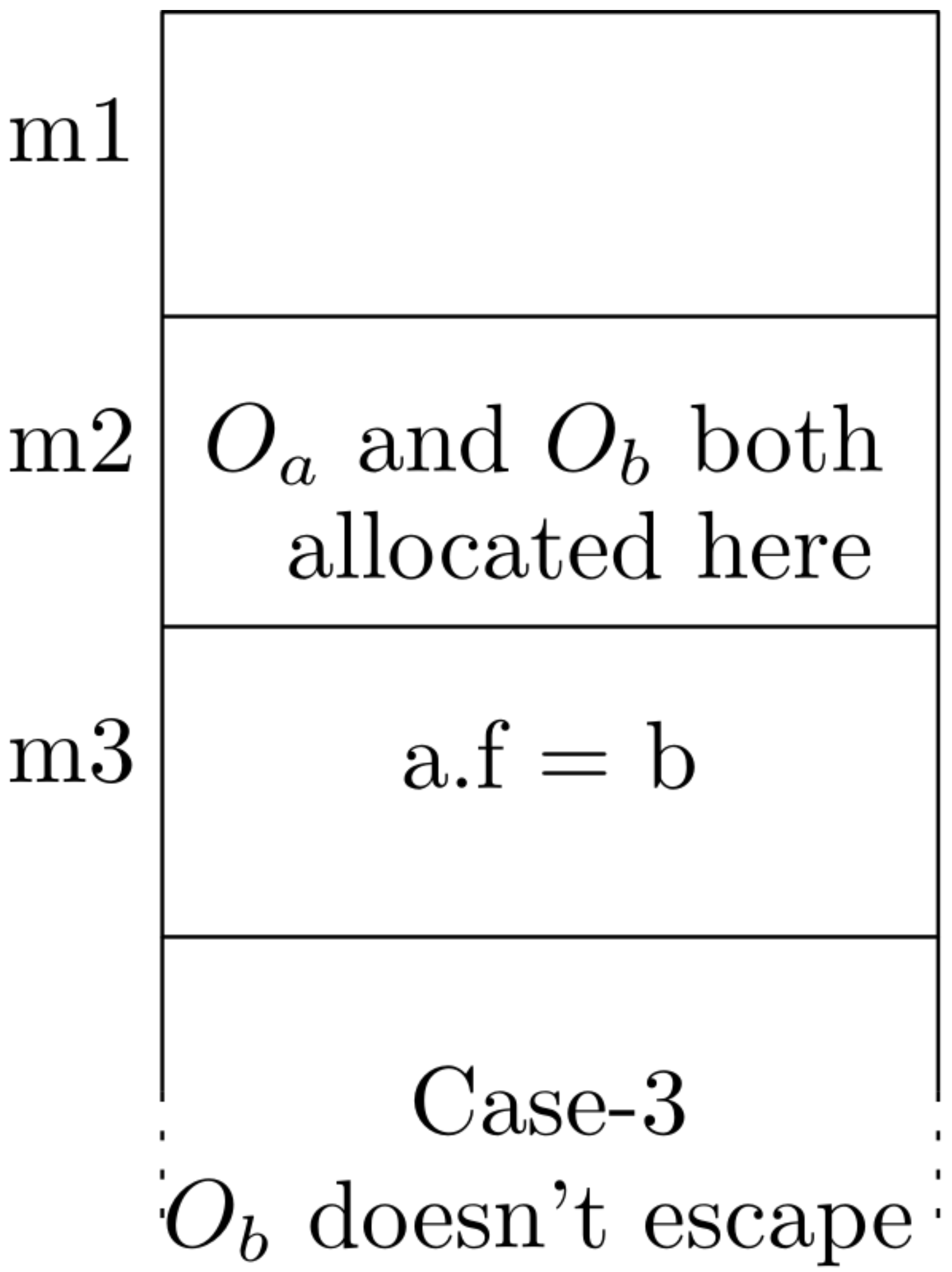
Ordering Objects on Stack

- A simple address-comparison check works majority of times.



Ordering Objects on Stack

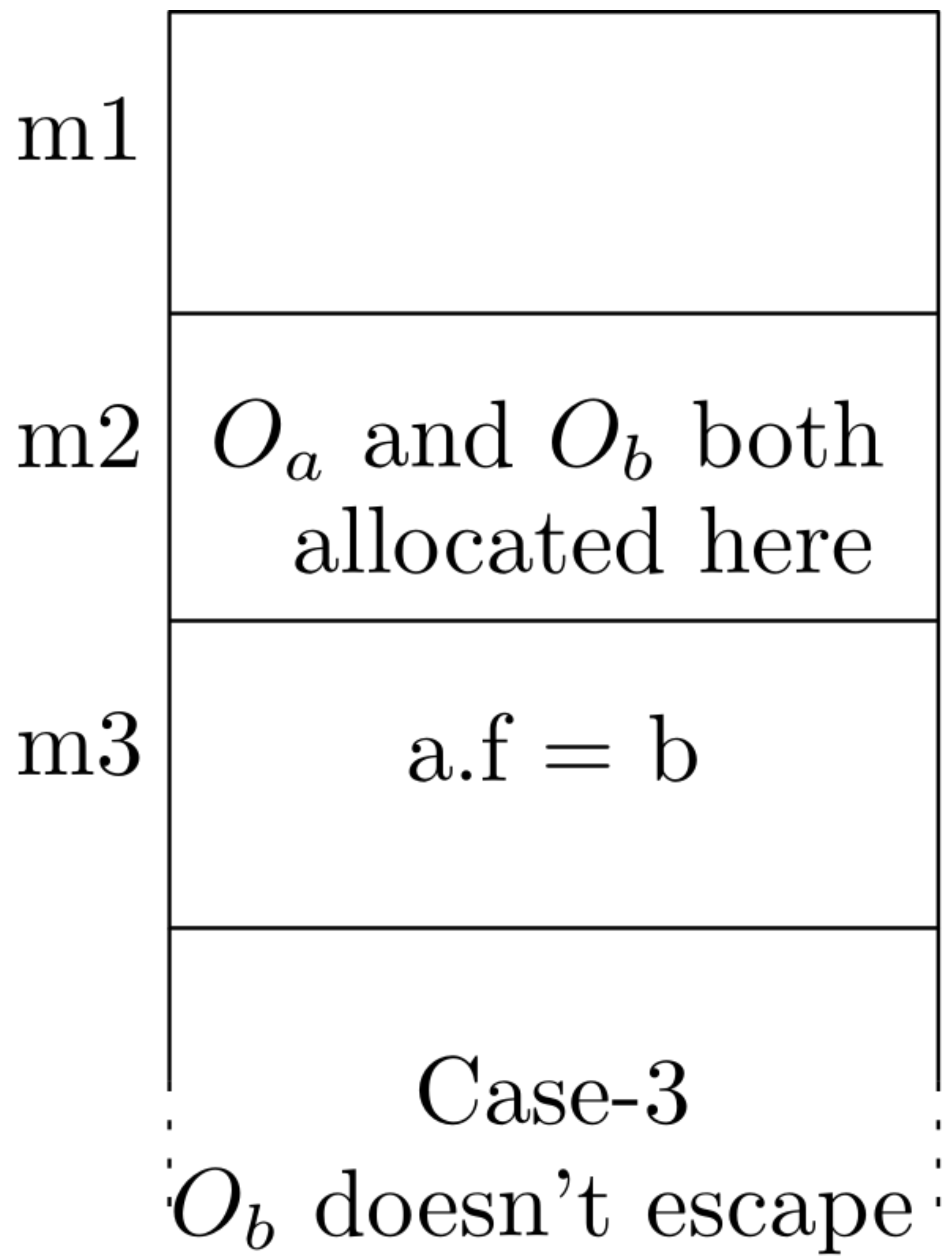
- A simple address-comparison check works majority of times.



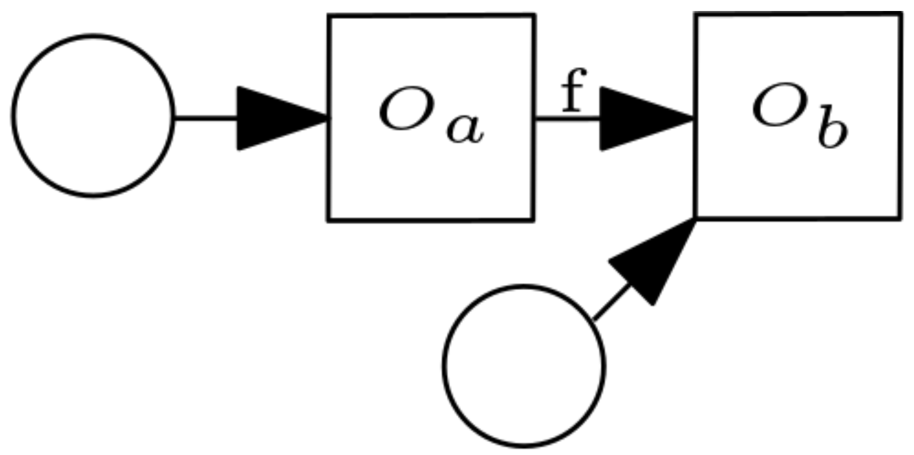
- Statically create a partial order of stack-allocatable objects.

Ordering Objects on Stack

- A simple address-comparison check works majority of times.

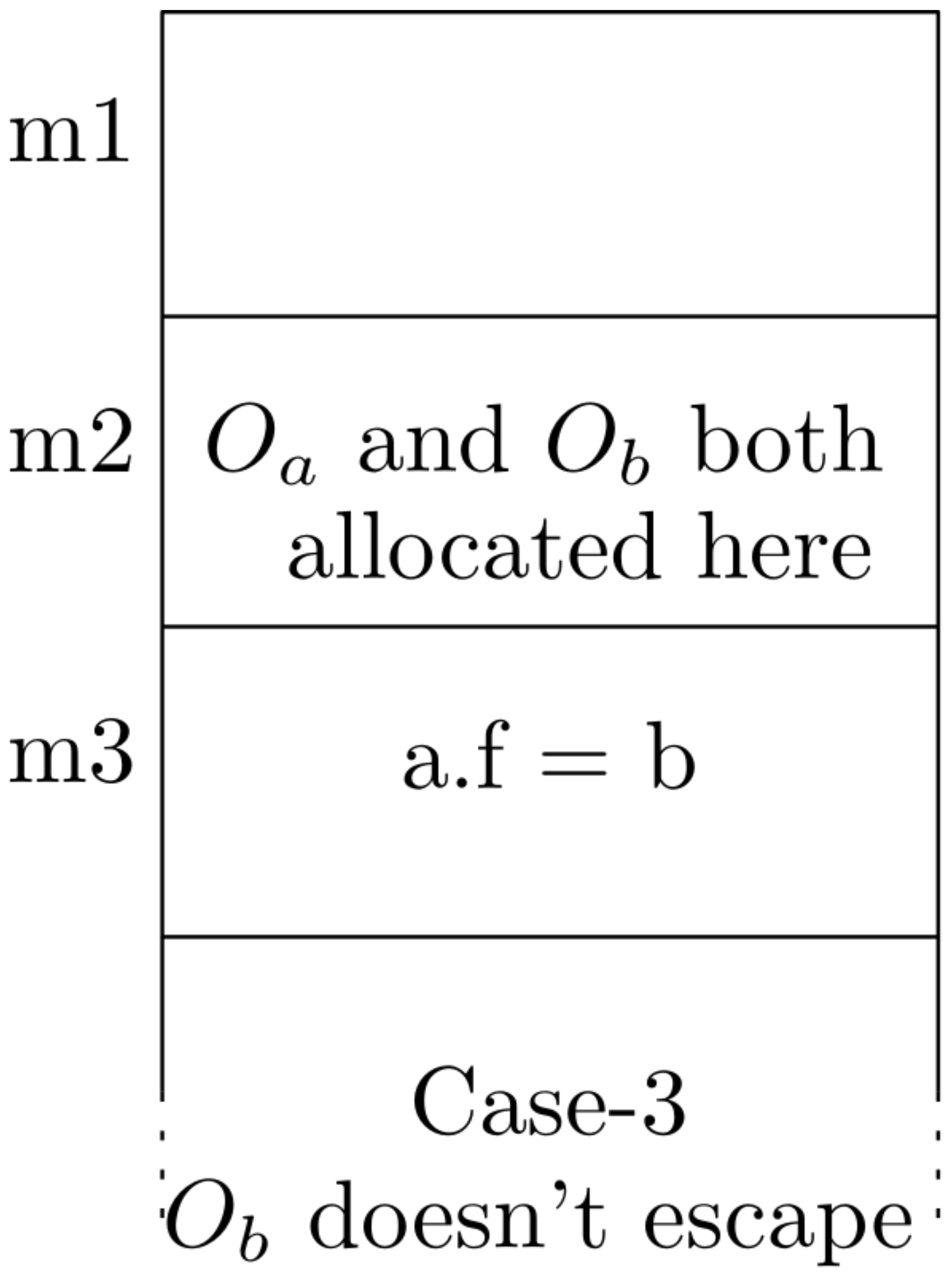


- Statically create a partial order of stack-allocatable objects.

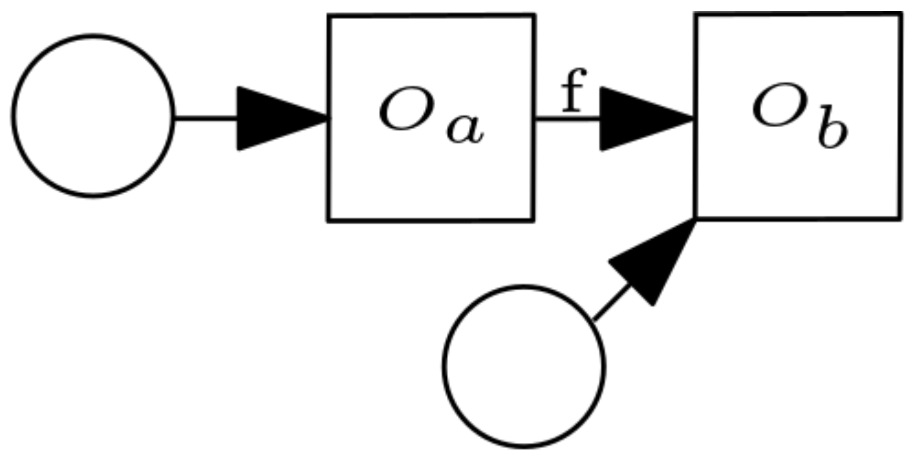


Ordering Objects on Stack

- A simple address-comparison check works majority of times.



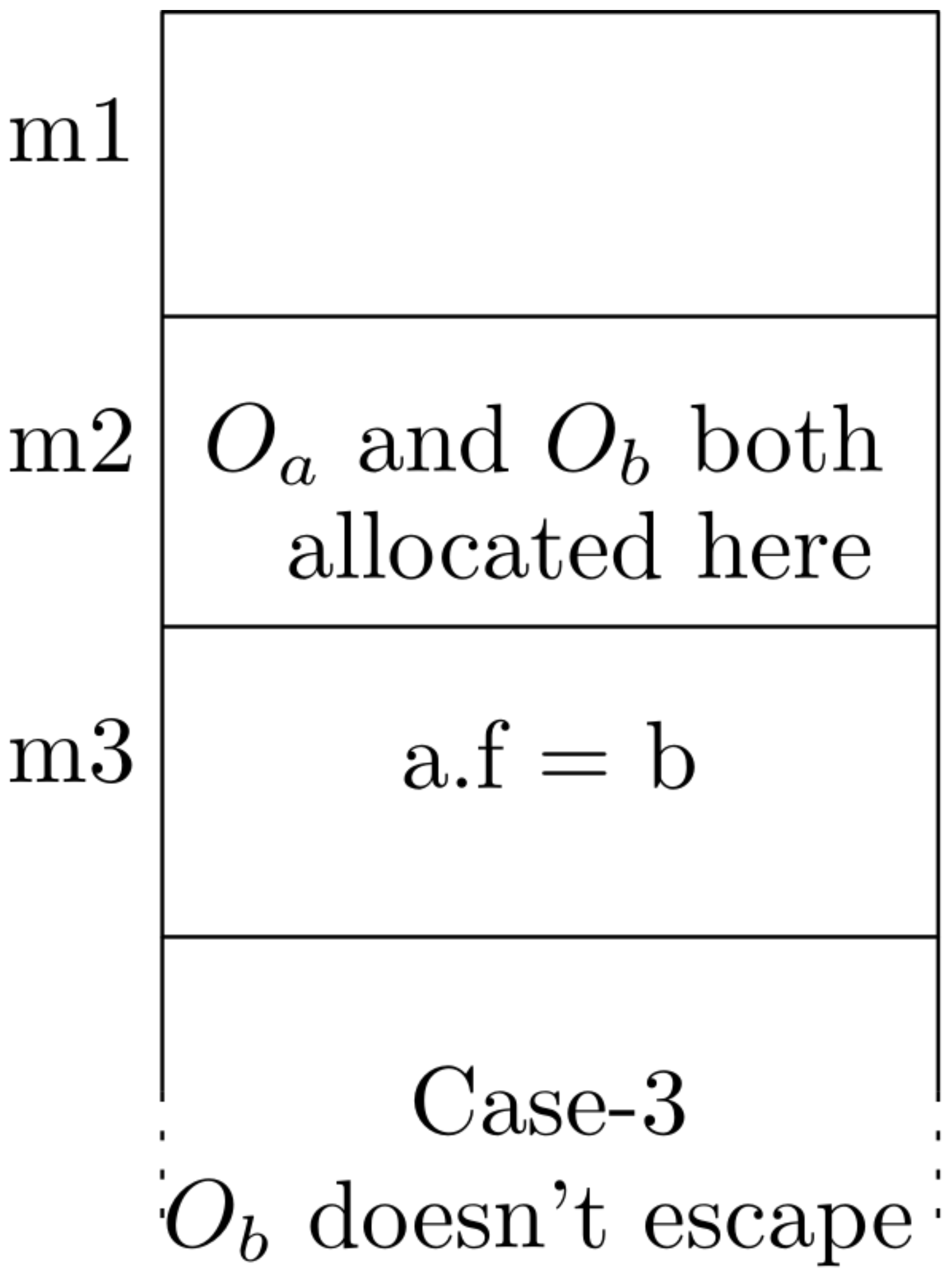
- Statically create a partial order of stack-allocatable objects.



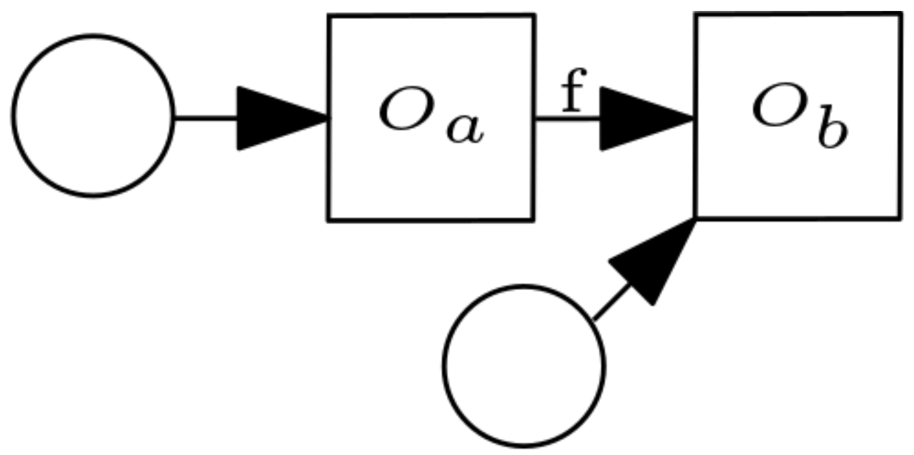
$[O_b, O_a]$

Ordering Objects on Stack

- A simple address-comparison check works majority of times.



- Statically create a partial order of stack-allocatable objects.

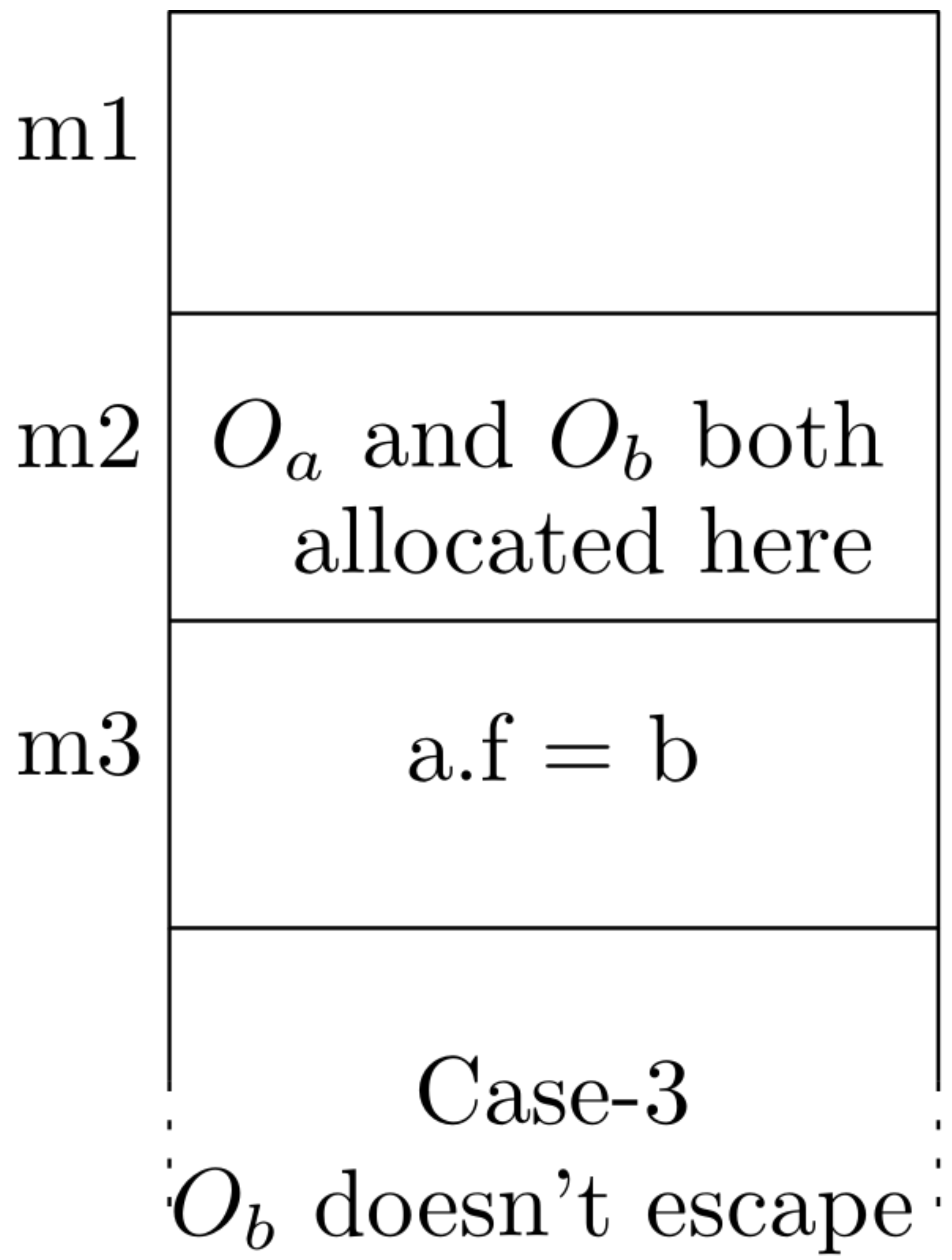


$[O_b, O_a]$

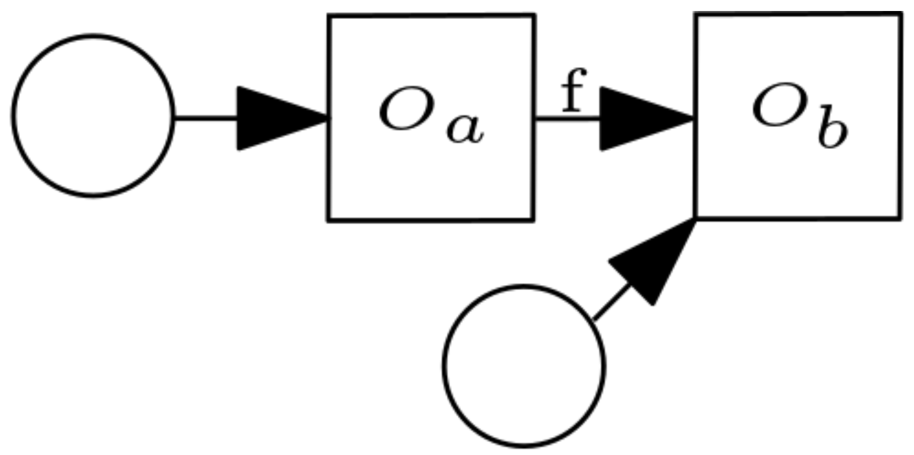
- Use the stack-order in VM to re-order the list of stack allocated objects.

Ordering Objects on Stack

- A simple address-comparison check works majority of times.



- Statically create a partial order of stack-allocatable objects.

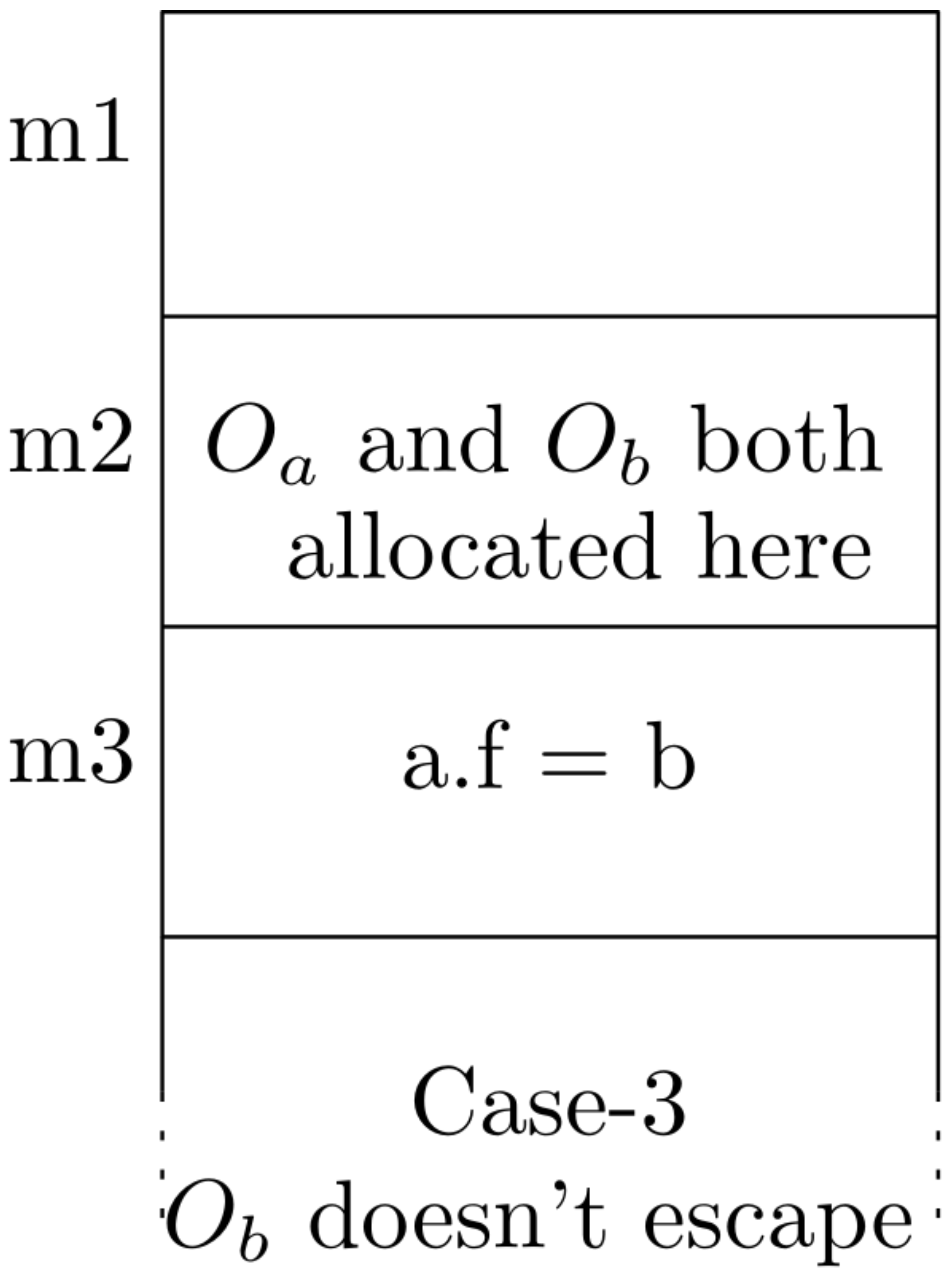


$[O_b, O_a]$

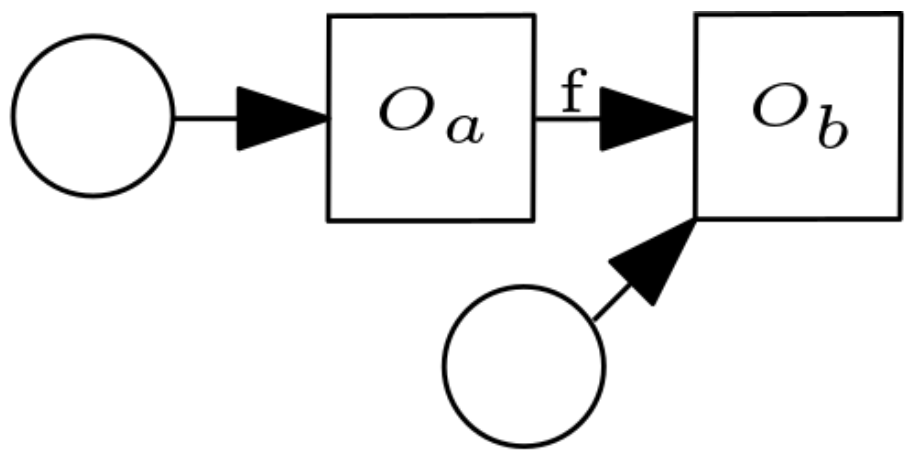
- Use the stack-order in VM to re-order the list of stack allocated objects.
- Reduces cost of heapification checks.

Ordering Objects on Stack

- A simple address-comparison check works majority of times.



- Statically create a partial order of stack-allocatable objects.

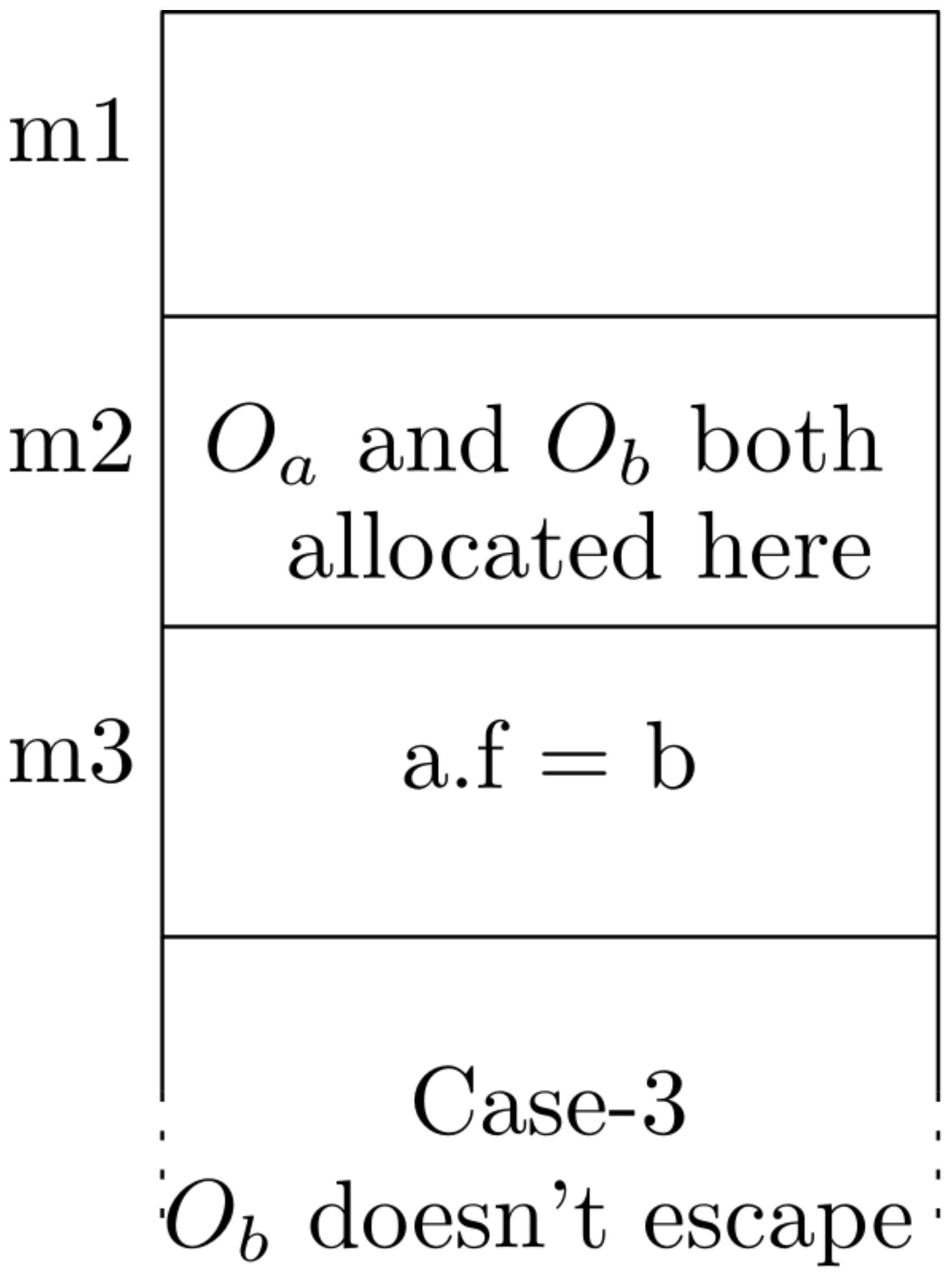


$[O_b, O_a]$

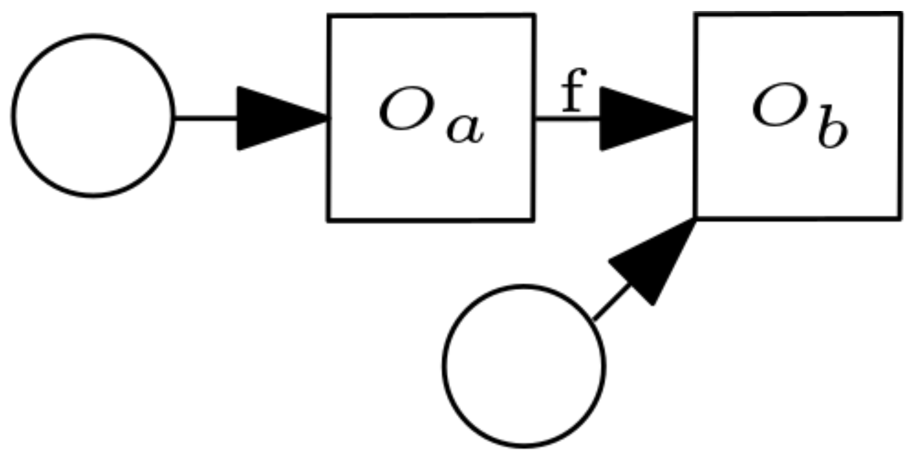
- Use the stack-order in VM to re-order the list of stack allocated objects.
- Reduces cost of heapification checks.
- In case of cycles — result will not be valid only for one store statement.

Ordering Objects on Stack

- A simple address-comparison check works majority of times.



- Statically create a partial order of stack-allocatable objects.



$[O_b, O_a]$

- Use the stack-order in VM to re-order the list of stack allocated objects.
- Reduces cost of heapification checks.
- In case of cycles — result will not be valid only for one store statement. Stack Walk

Implementation and Evaluation

Implementation and Evaluation

- Implementation:
  - Static analysis: Soot
  - Runtime components: OpenJ9 VM

Implementation and Evaluation

- Implementation:
  - Static analysis: Soot
  - Runtime components: OpenJ9 VM
- Benchmarks:
  - DaCapo suites 23.10-chopin and 9.12 MRI.
  - SPECjvm 2008.

Implementation and Evaluation

- Implementation:
  - Static analysis: Soot
  - Runtime components: OpenJ9 VM
- Benchmarks:
  - DaCapo suites 23.10-chopin and 9.12 MRI.
  - SPECjvm 2008.
- Evaluation schemes:
  - **BASE**: Stack allocation with the existing scheme.
  - **OPT**: Stack allocation with our optimistic scheme.

Implementation and Evaluation

- Implementation:
  - Static analysis: Soot
  - Runtime components: OpenJ9 VM
- Benchmarks:
  - DaCapo suites 23.10-chopin and 9.12 MRI.
  - SPECjvm 2008.
- Evaluation schemes:
  - **BASE**: Stack allocation with the existing scheme.
  - **OPT**: Stack allocation with our optimistic scheme.
- Compute:
  - Enhancement in stack allocation.
  - Impact on performance and garbage collection.

Evaluation (Stack Allocation)

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Evaluation (Stack Allocation)

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Evaluation (Stack Allocation)

	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	**Static Count**	**Dynamic Count**	**Stack Bytes**	**Static Count**	**Dynamic Count**	**Stack Bytes**
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Evaluation (Stack Allocation)

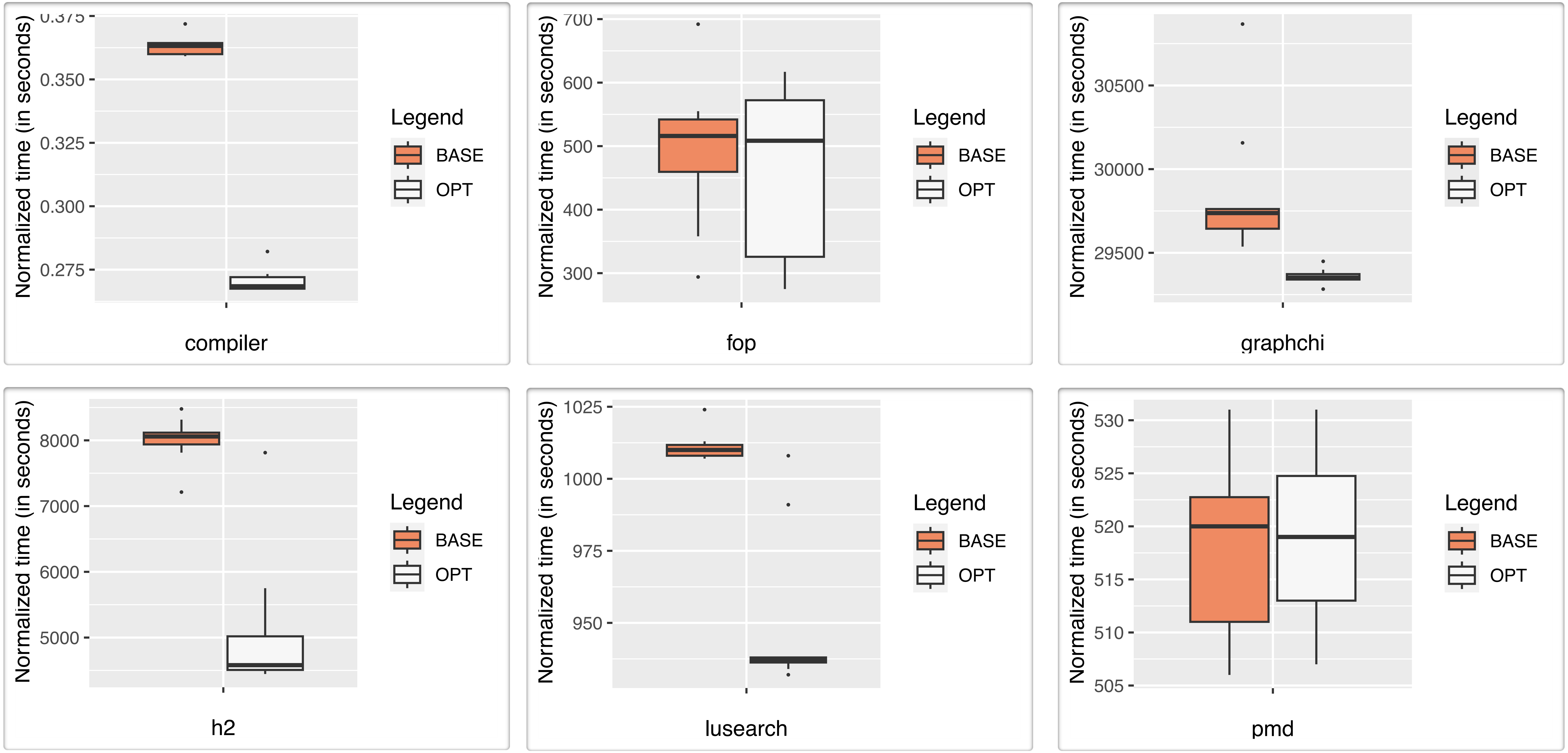
	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	523MB	94 (3.87%)	452M (13.92%)	10801MB
luindex	35 (1.35%)	3M (2.39%)	98MB	89 (3.49%)	5M (3.49%)	133MB
lusearch	30 (1.09%)	25M (3.23%)	775MB	78 (3.05%)	59M (7.4%)	1686MB
pmd	89 (1.09%)	52M (7.20%)	1310MB	191 (3.97%)	105M (14.2%)	2465MB
compiler	93 (1.73%)	94M (5.50%)	1720MB	137 (2.75%)	105M (6.17%)	2329MB
rsa	16 (1.13%)	0.1M (1.1%)	46MB	35 (3.18%)	7M (4.62%)	170MB
signverify	15 (0.84%)	0.24M (0.86%)	6.8MB	51 (3.10%)	2.1M (7.24%)	49.4MB

Evaluation (Stack Allocation)

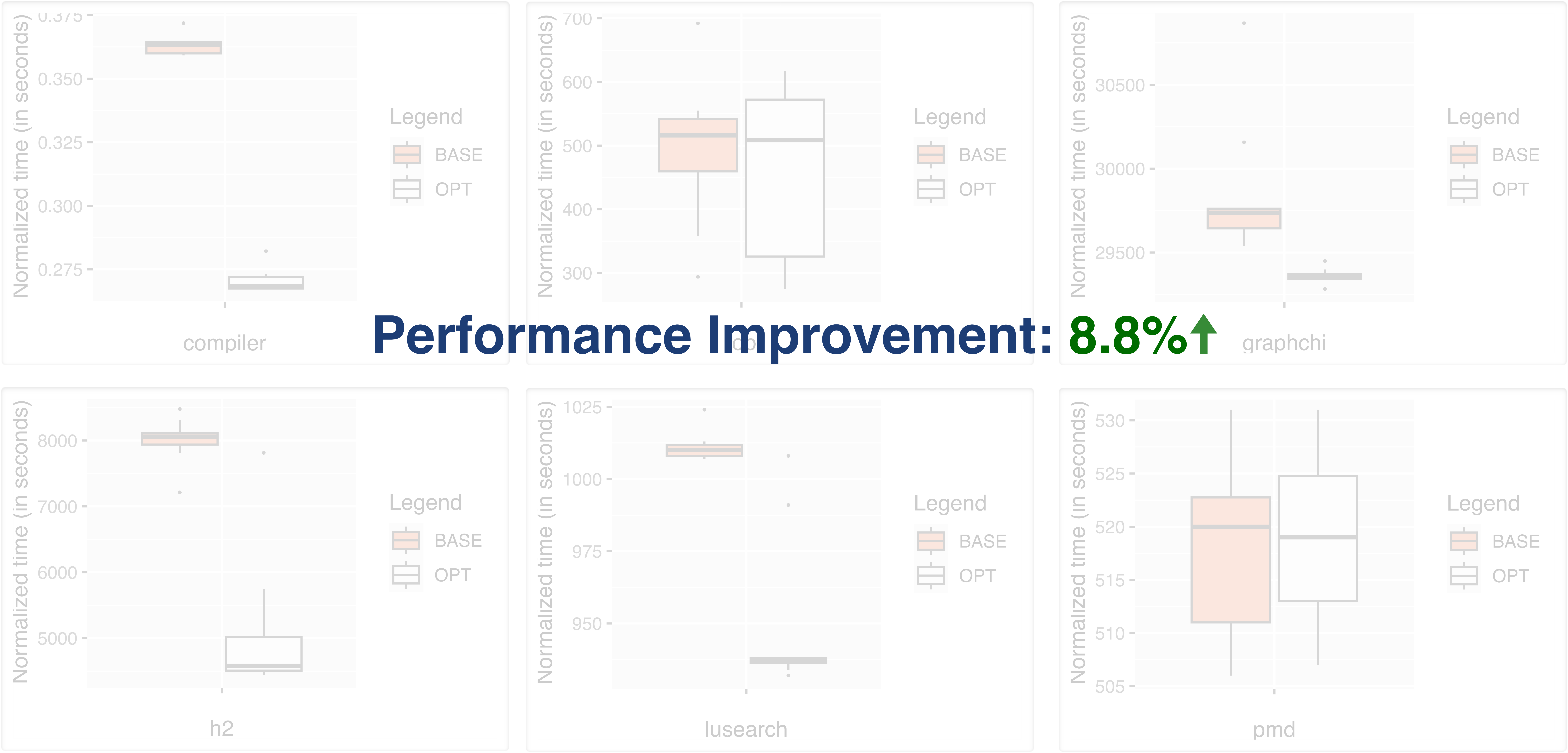
Stack Allocation: 71%↑ Stack Bytes: 54%↑

(Less Heap Allocation)

Performance

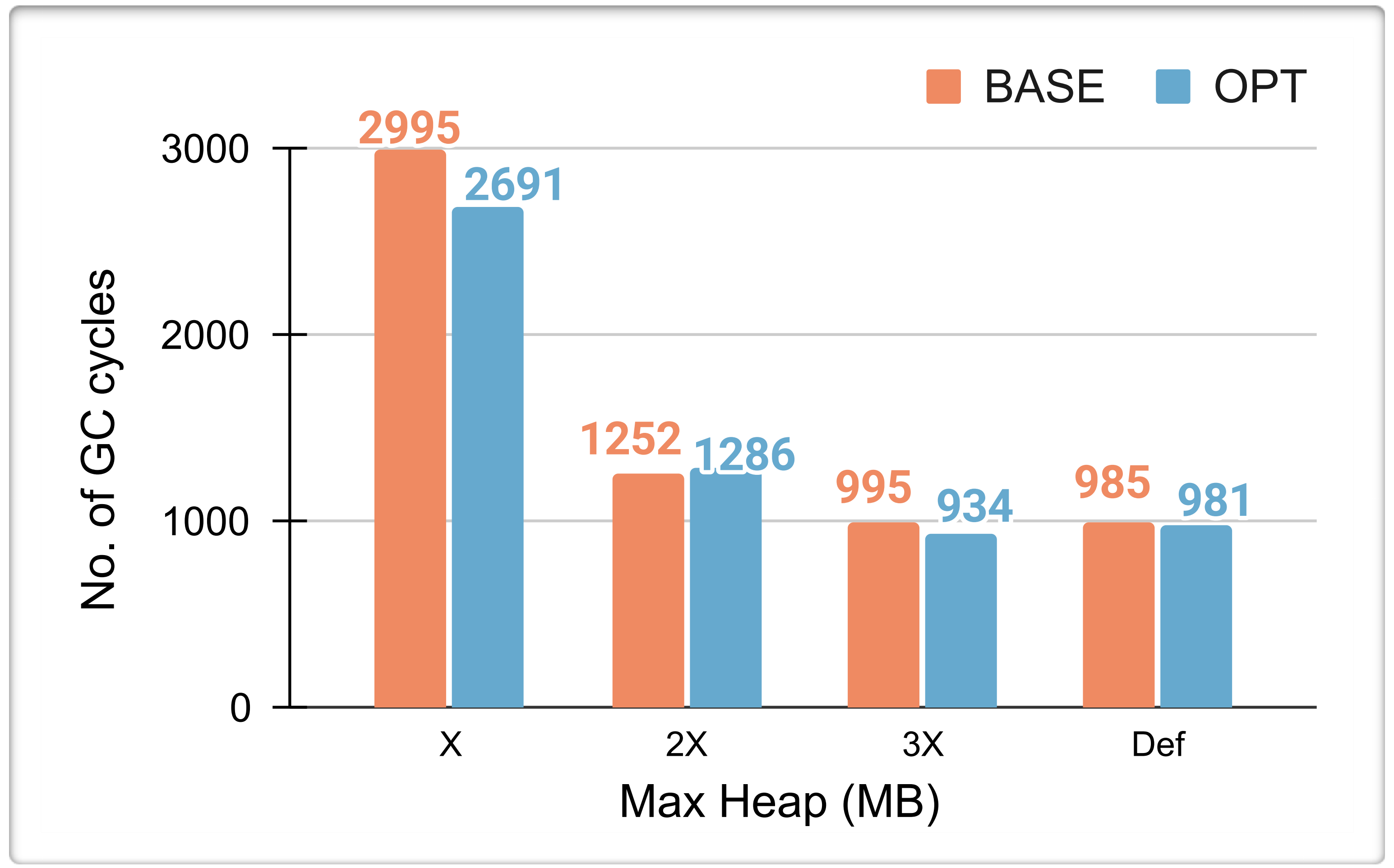


Performance

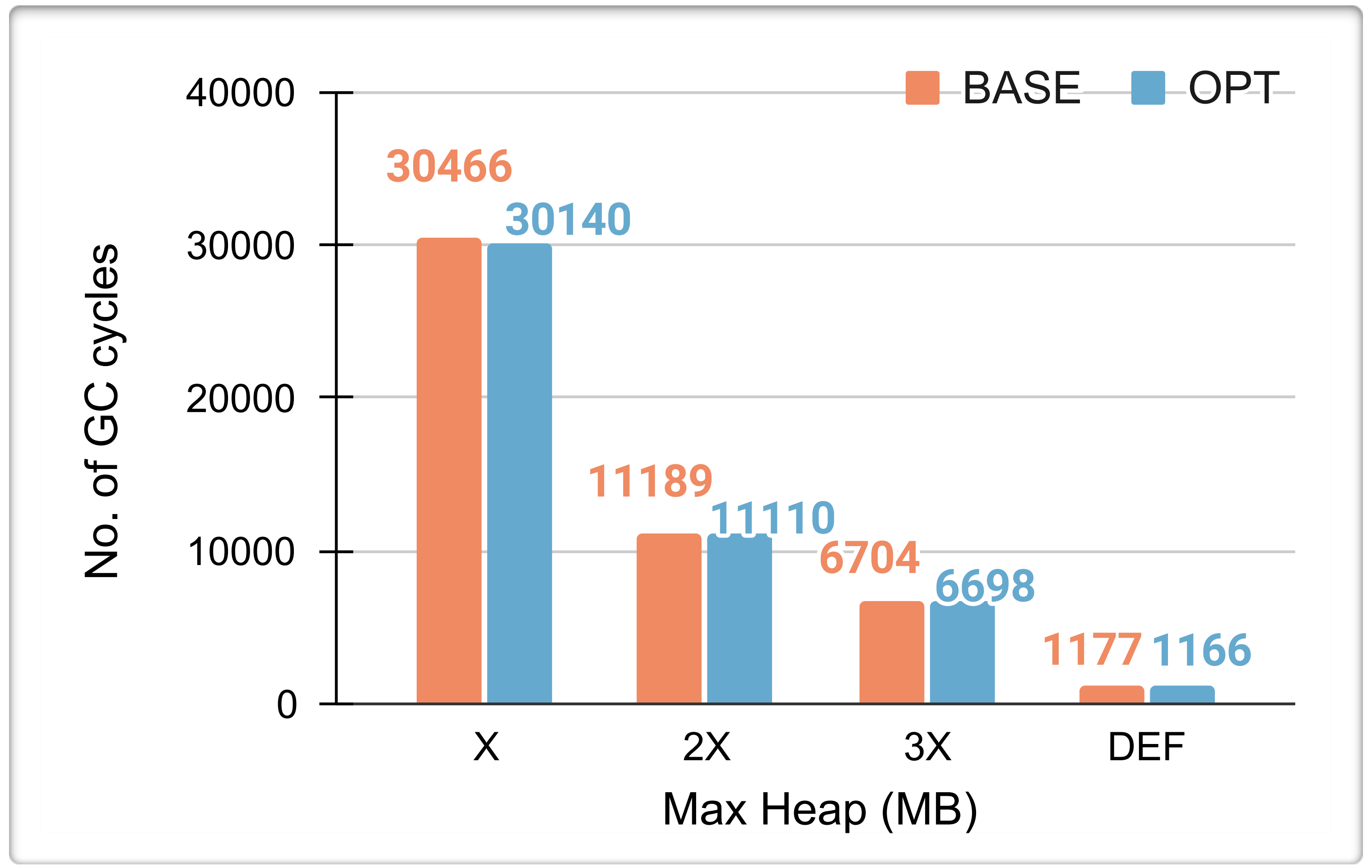


Performance Improvement: 8.8%↑

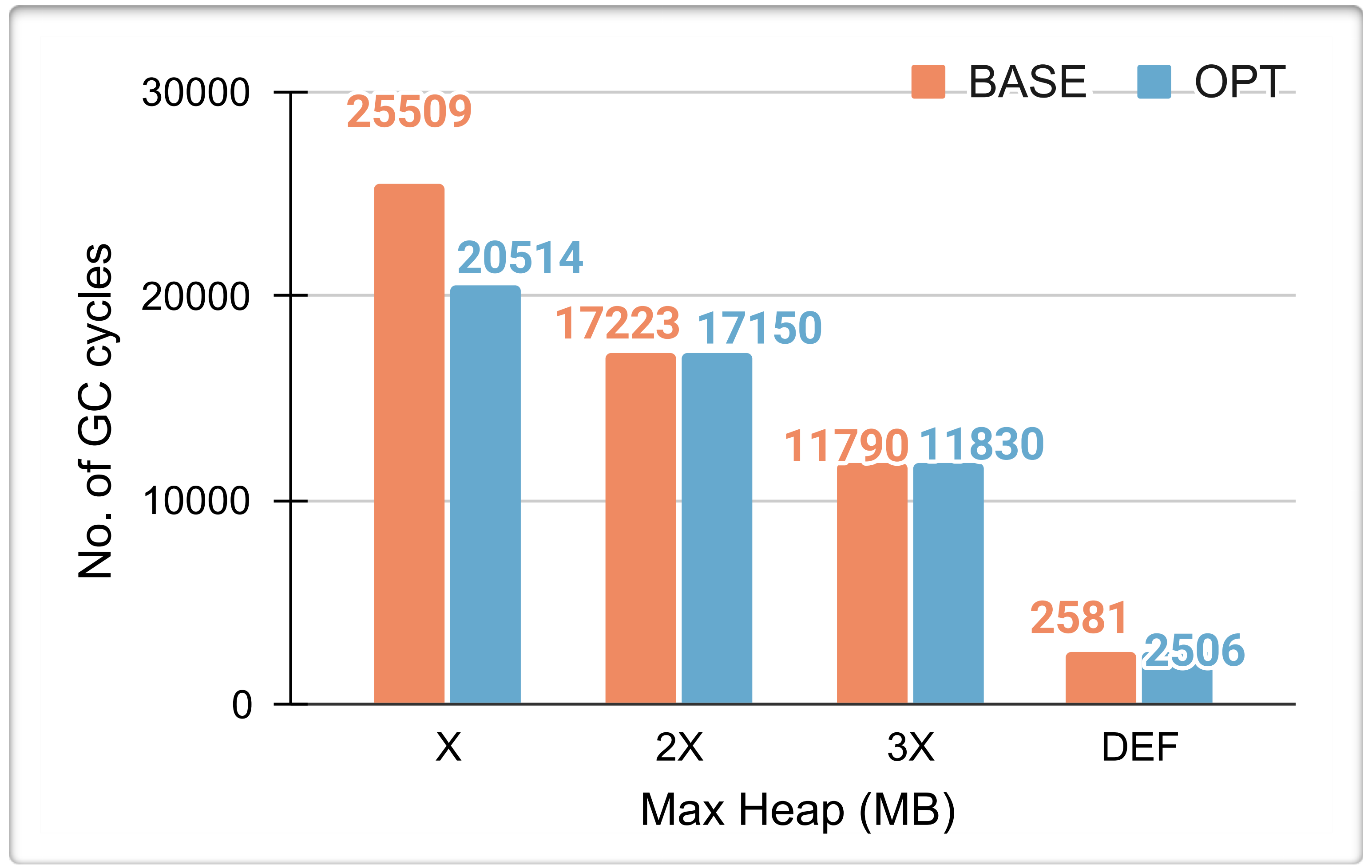
Garbage Collection



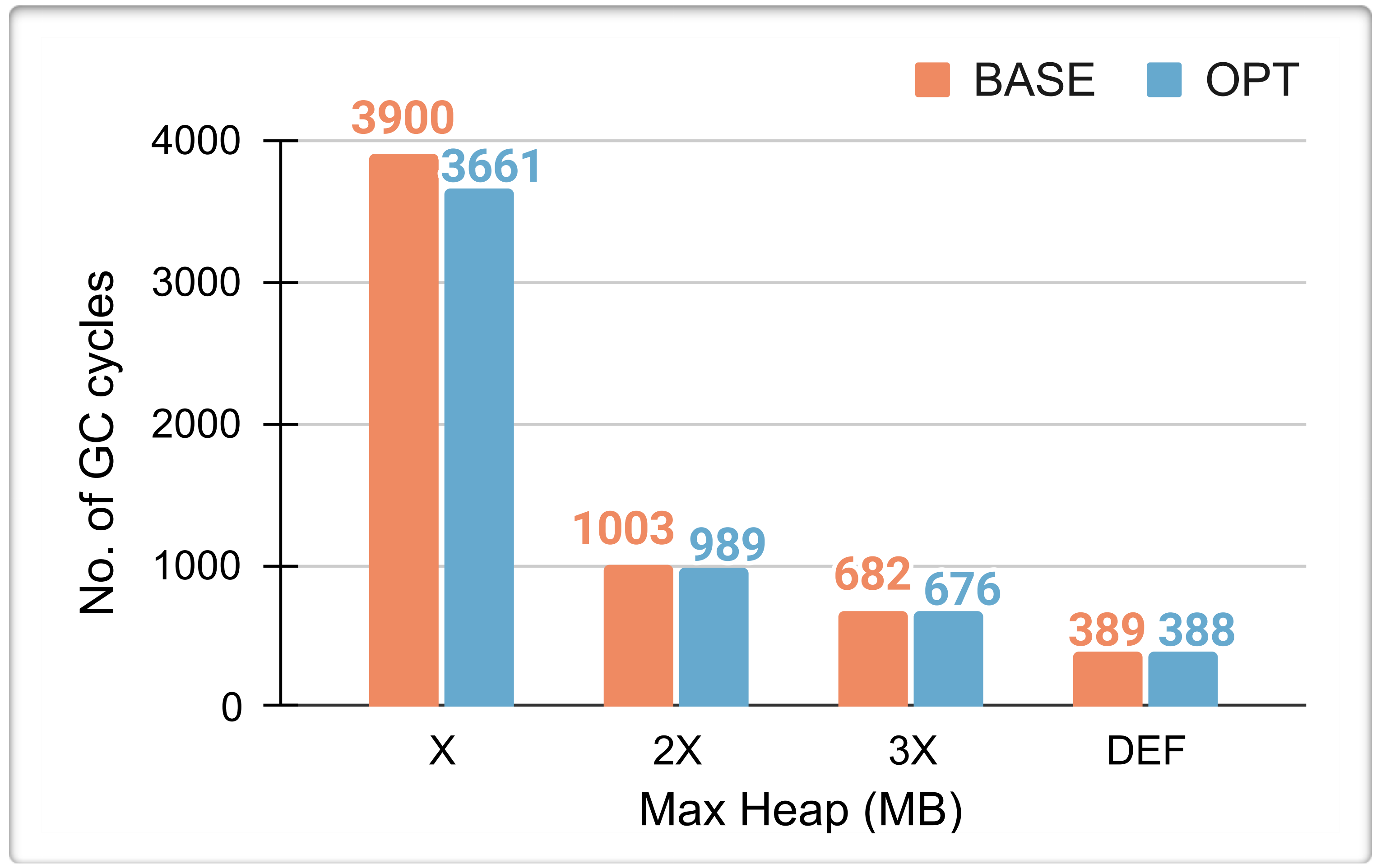
compiler



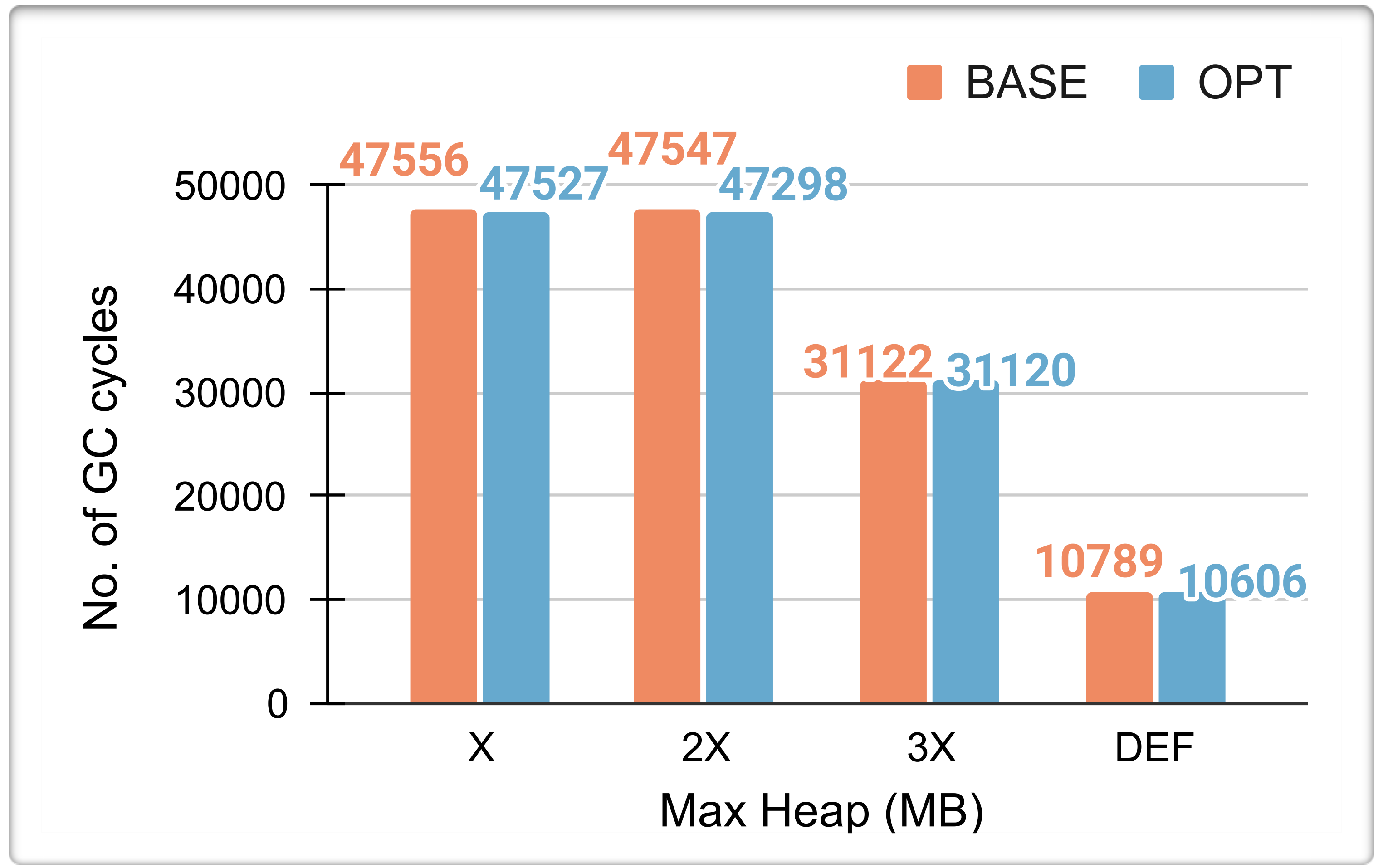
fop



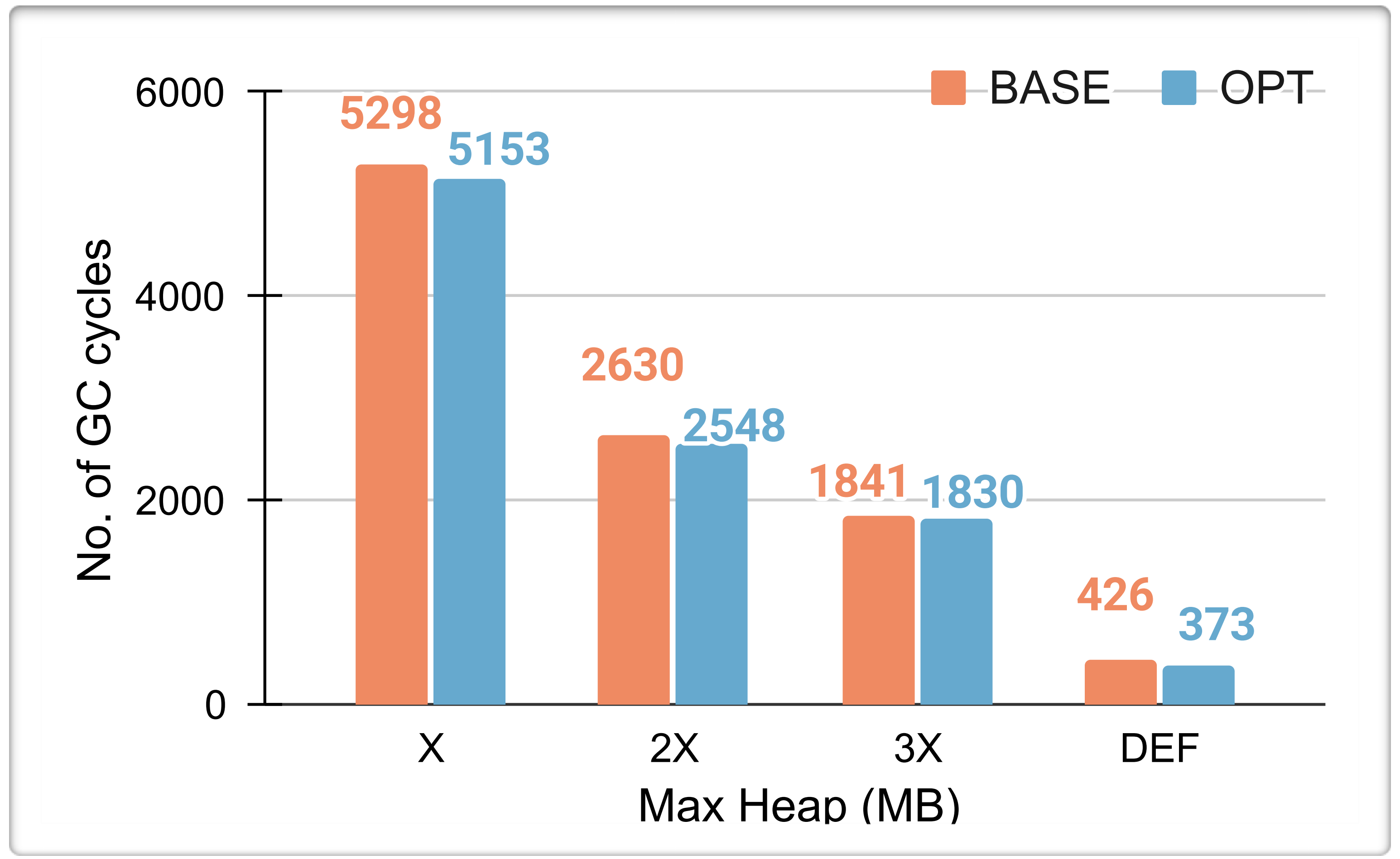
graphchi



h2

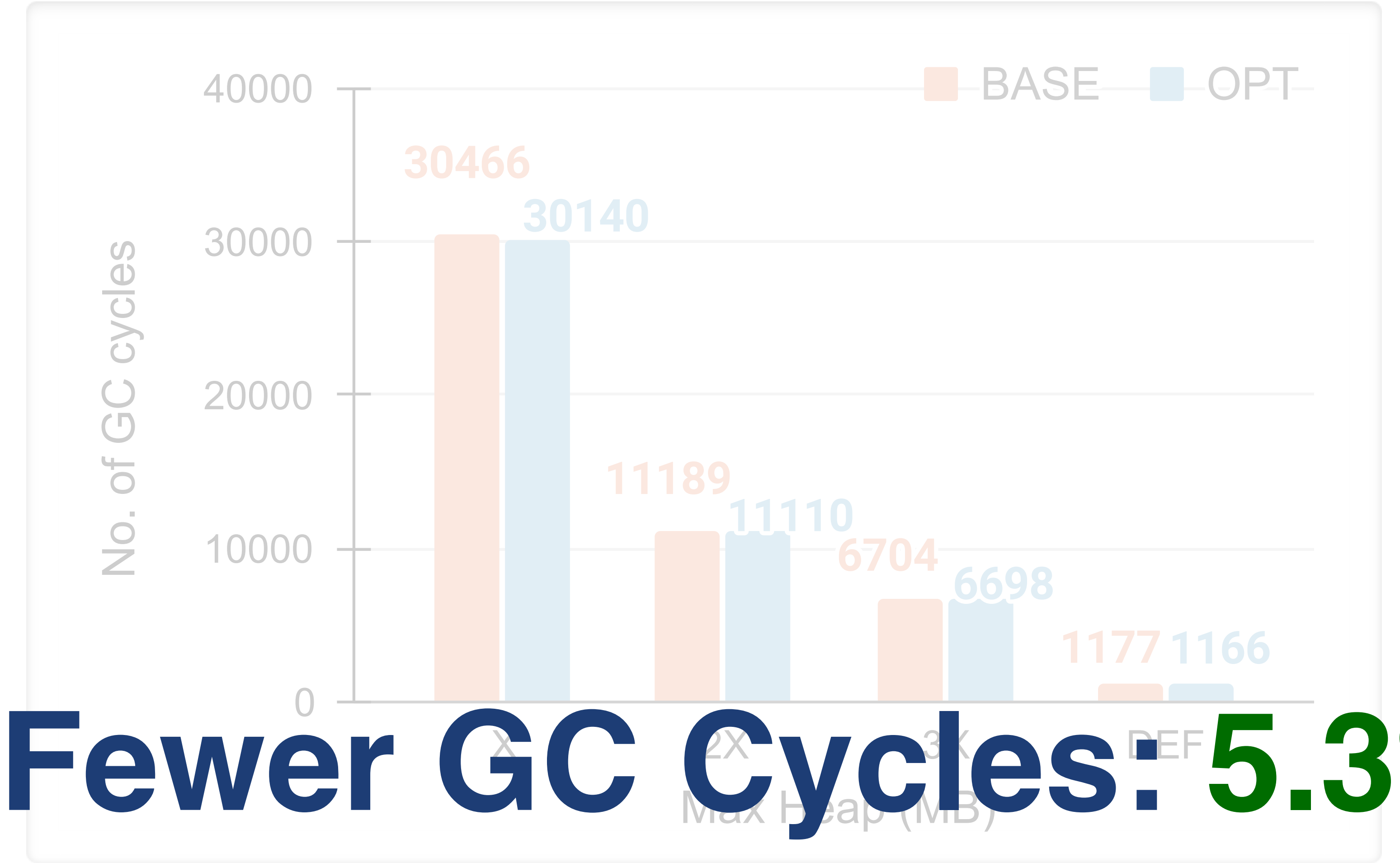


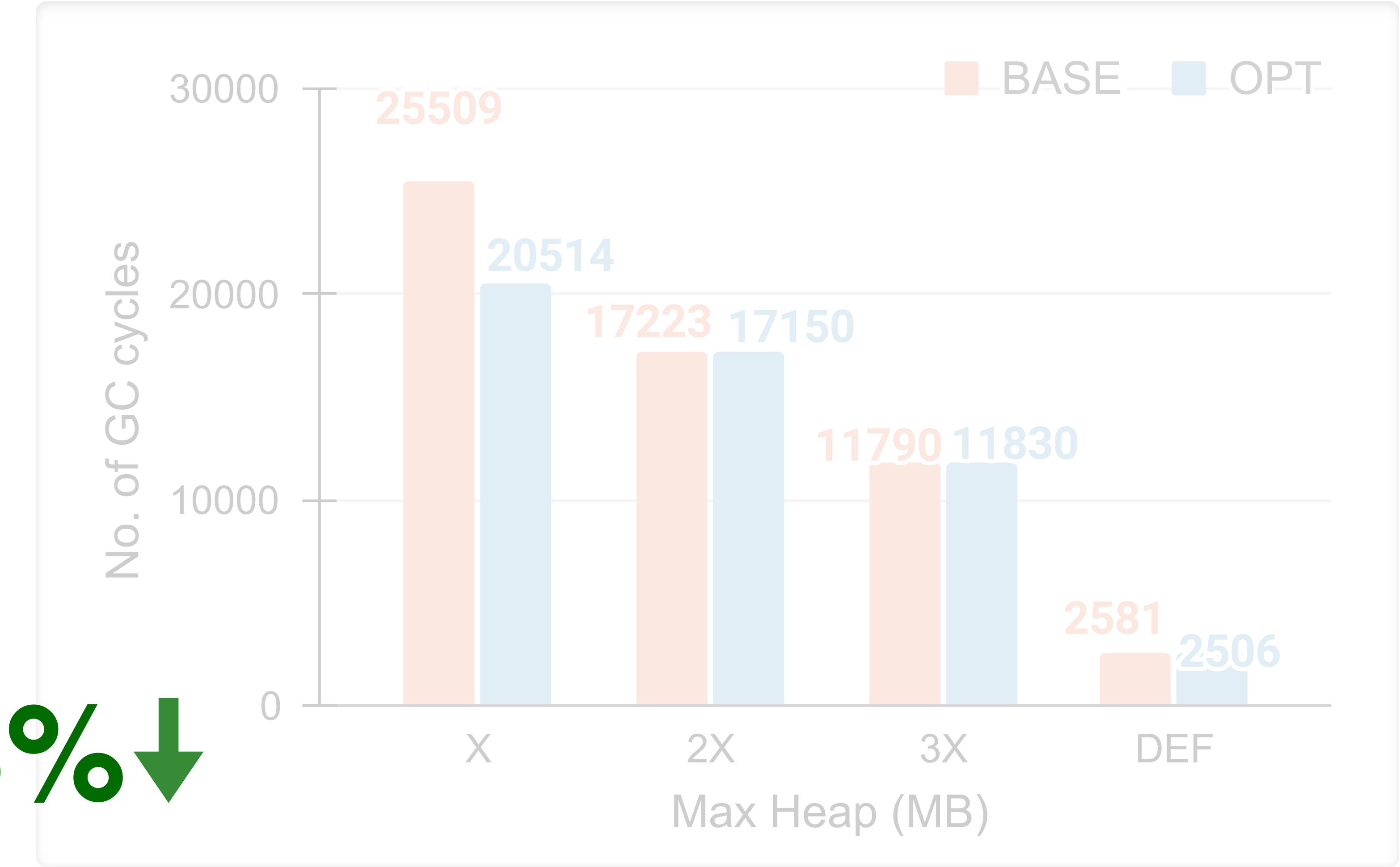
lusearch



pmd

Garbage Collection

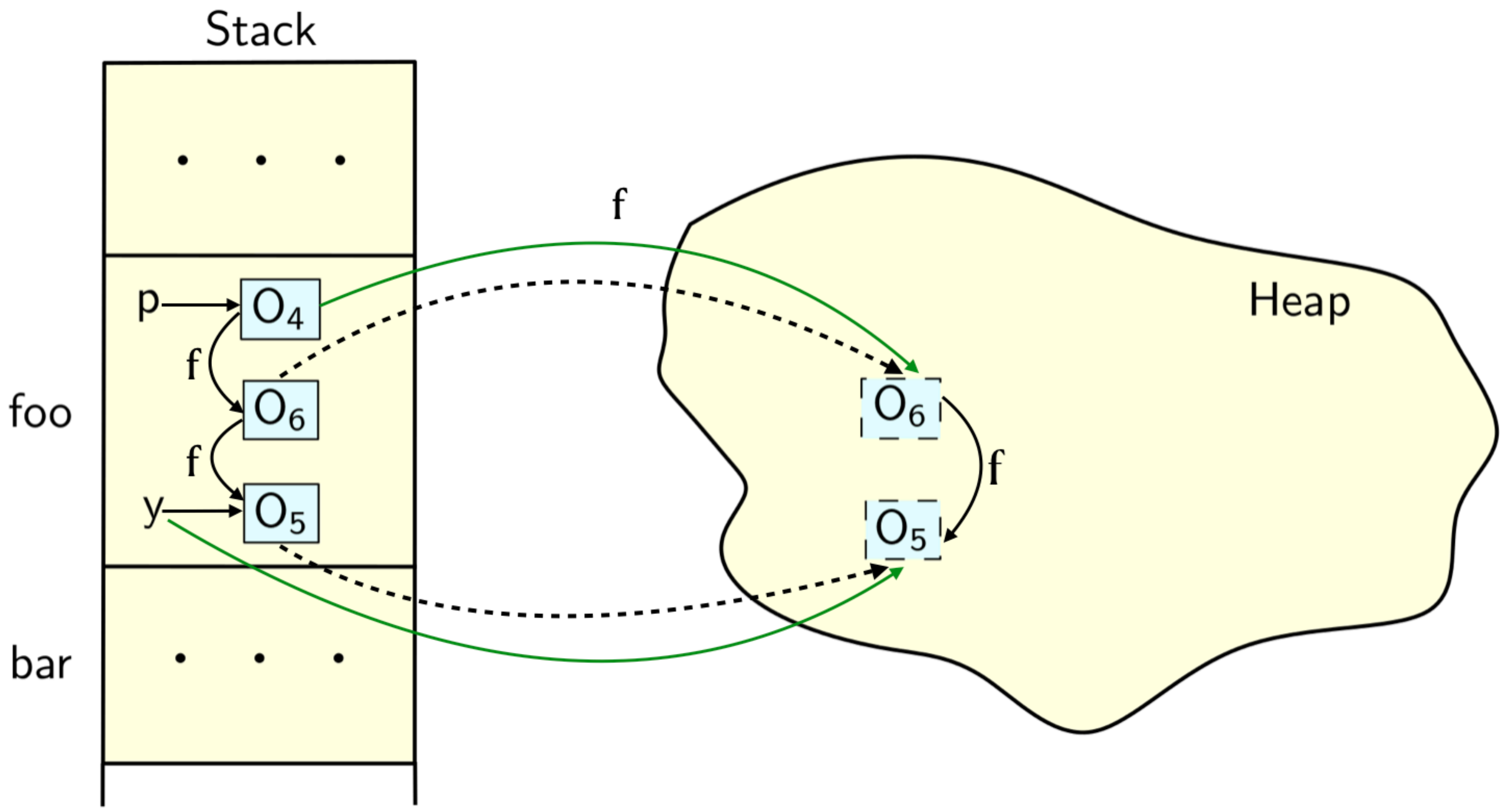


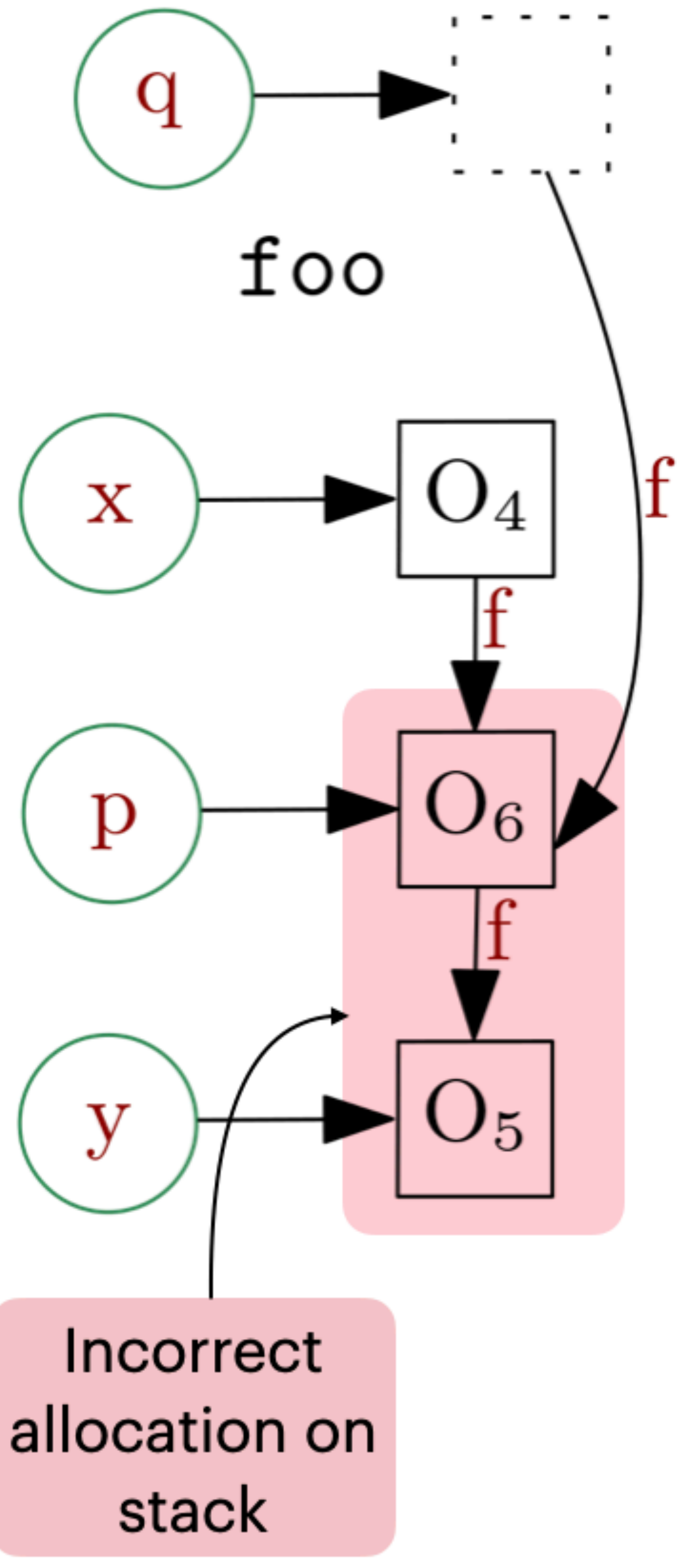


Fewer GC Cycles: 5.3%↓

Take Aways

- An important OO Optimization: Allocating method-local objects on the stack frames of their allocating methods.
- Used static escape analysis to optimistically allocate identified objects on stack to improve the precision without thwarting the efficiency.
- Ensure functional correctness in cases static analysis results do not correspond to the runtime environment.





- **Overall, one of the first approaches to soundly and efficiently use static (offline) analysis results in a JIT compiler!**

Thank You!! Questions?

Take Aways

Take Aways

- An important OO Optimization: Allocating method-local objects on the stack frames of their allocating methods.

Take Aways

- An important OO Optimization: Allocating method-local objects on the stack frames of their allocating methods.

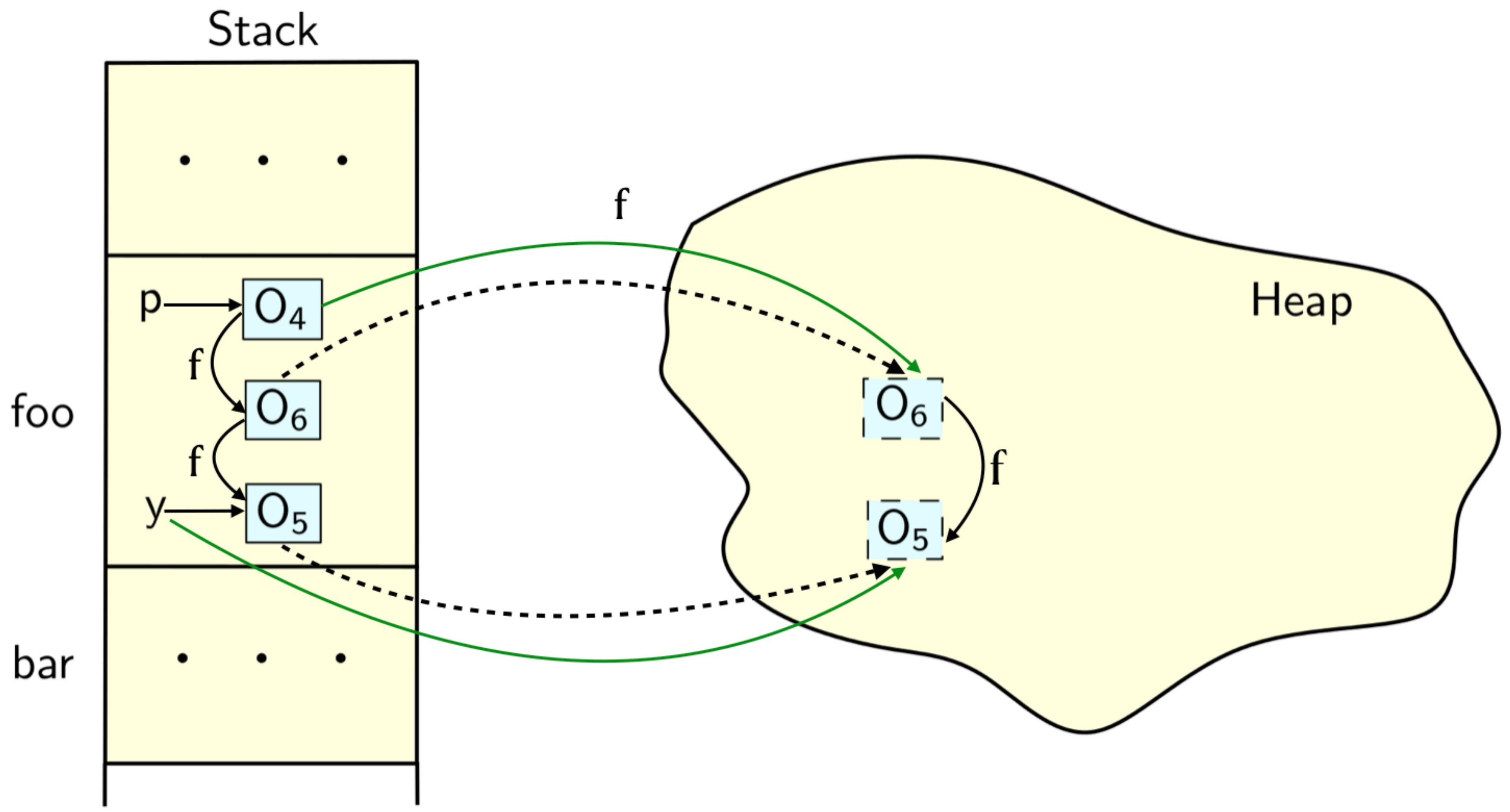
- Used static escape analysis to optimistically allocate identified objects on stack to improve the precision without thwarting the efficiency.

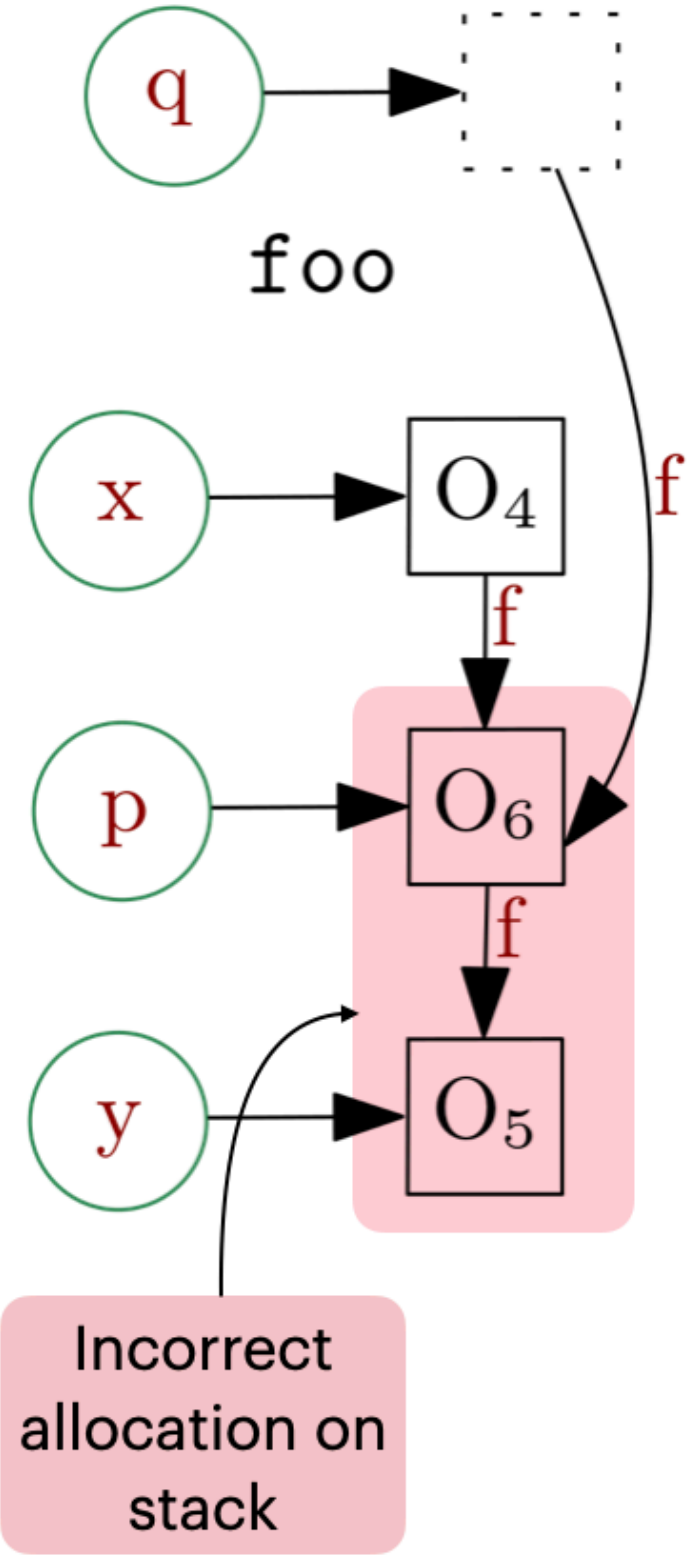
Take Aways

- An important OO Optimization: Allocating method-local objects on the stack frames of their allocating methods.

- Used static escape analysis to optimistically allocate identified objects on stack to improve the precision without thwarting the efficiency.

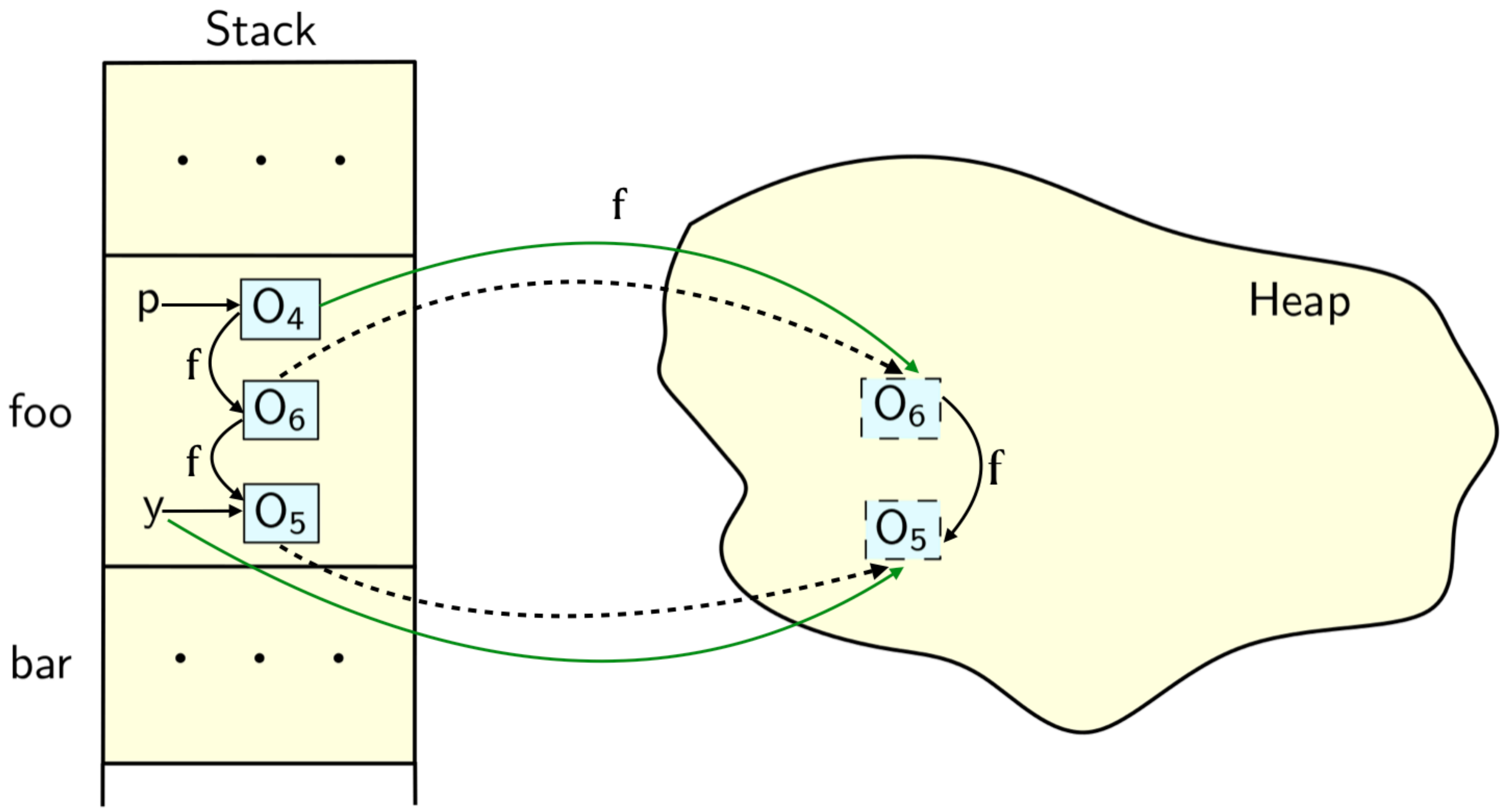
- Ensure functional correctness in cases static analysis results do not correspond to the runtime environment.

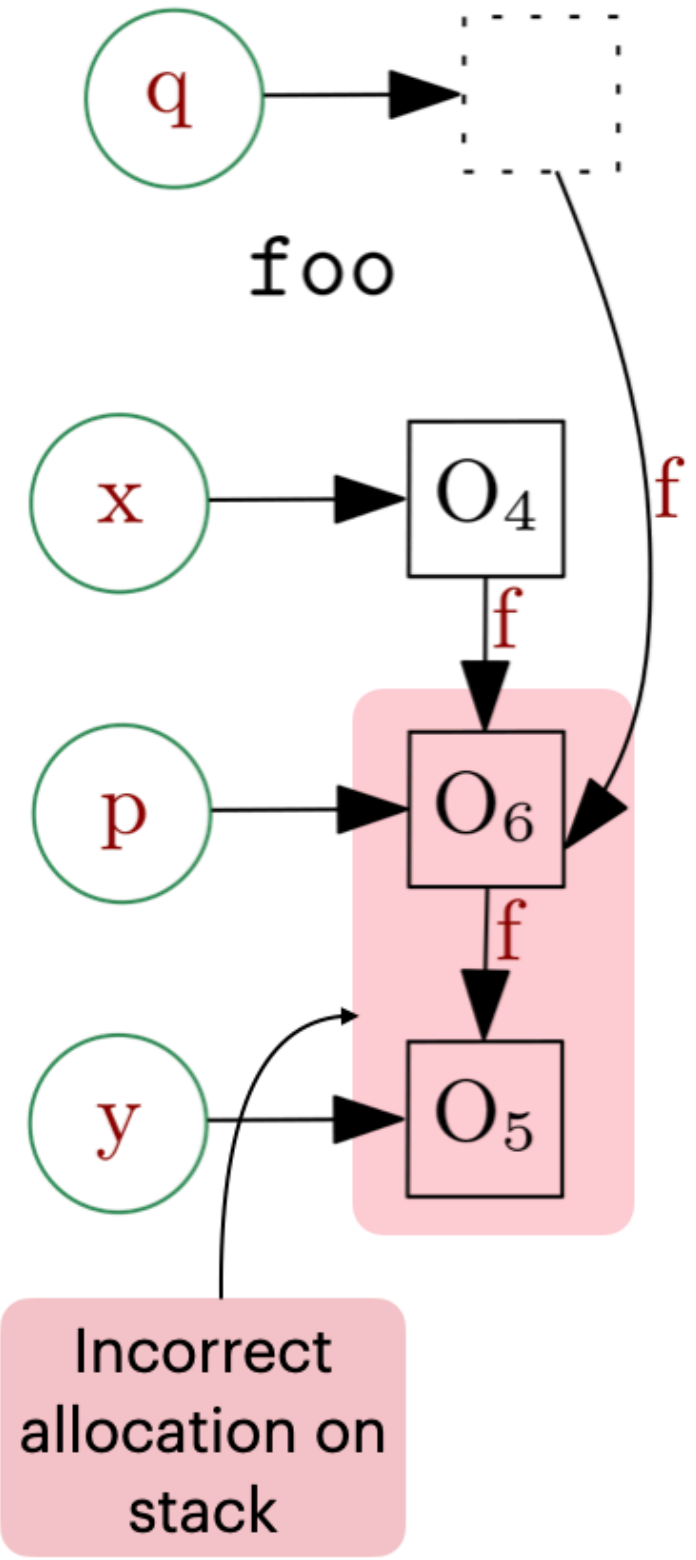




Take Aways

- An important OO Optimization: Allocating method-local objects on the stack frames of their allocating methods.
- Used static escape analysis to optimistically allocate identified objects on stack to improve the precision without thwarting the efficiency.
- Ensure functional correctness in cases static analysis results do not correspond to the runtime environment.





- **Overall, one of the first approaches to soundly and efficiently use static (offline) analysis results in a JIT compiler!**

Take Aways

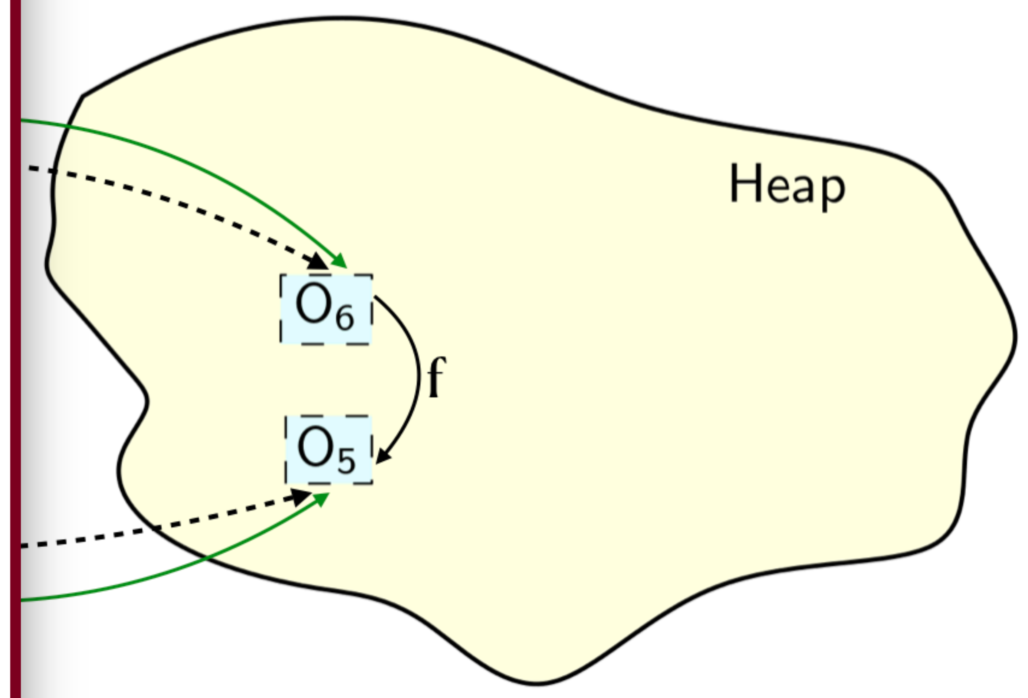
PLDI COPENHAGEN 2024

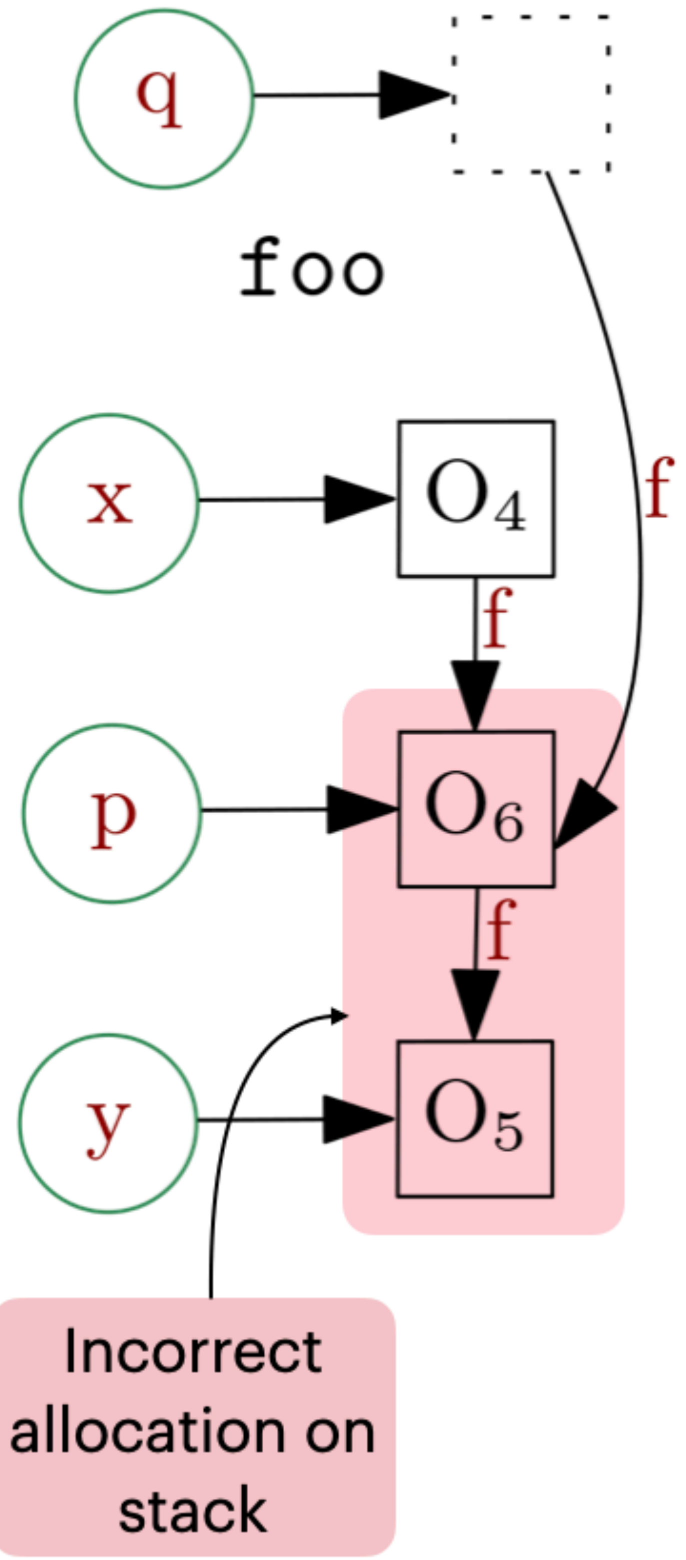
Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

tional correctness in
analysis results do not
to the runtime
t.





o **soundly** and **efficiently**
a **JIT compiler!**

Take Aways

PLDI COPENHAGEN 2024

Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

Paper Link

PLDI COPENHAGEN 2024

Take Aways

Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

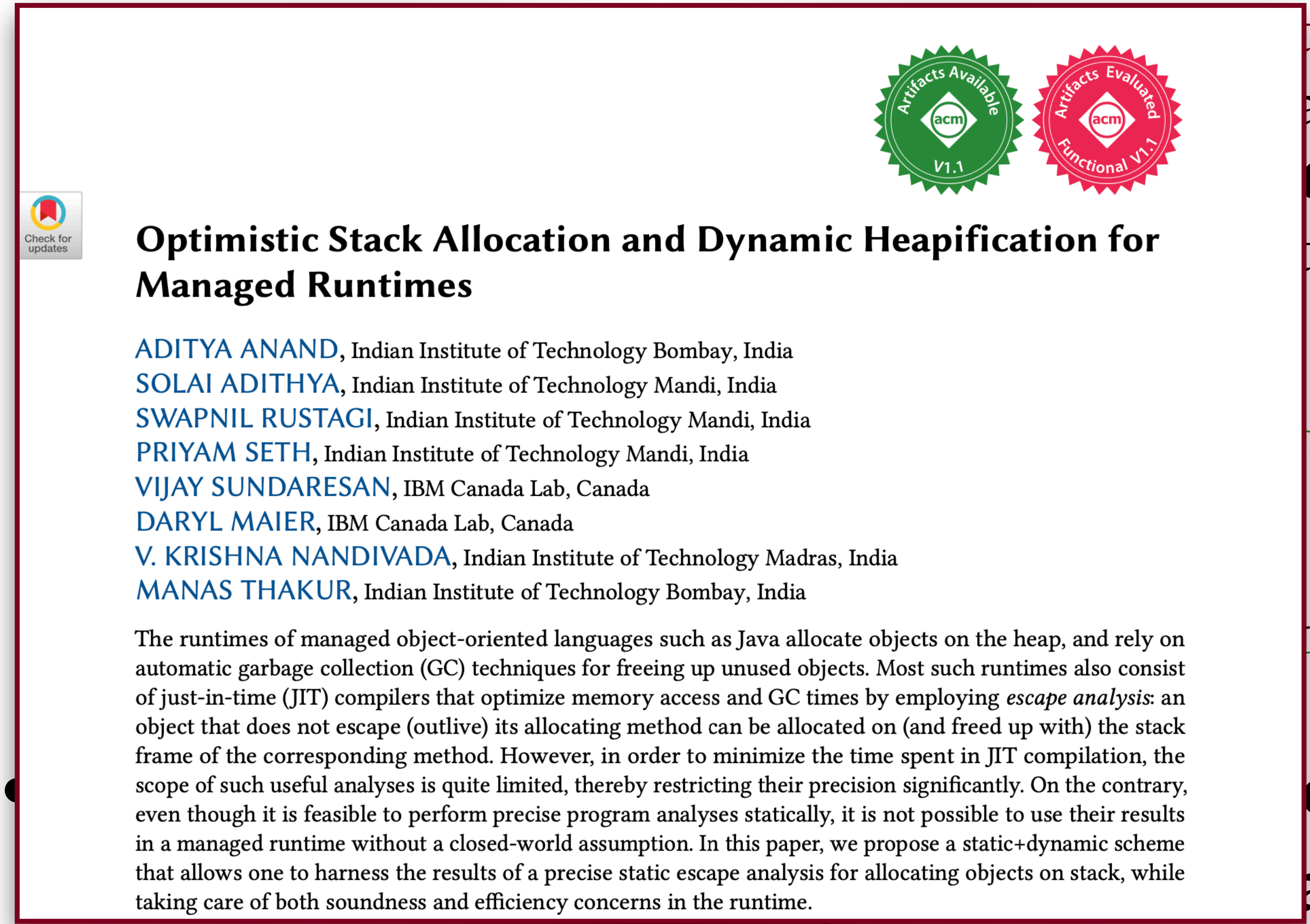
ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

Paper Link

Thank You!!

Take Aways



Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape analysis*: an object that does not escape (outlive) its allocating method can be allocated on (and freed up with) the stack frame of the corresponding method. However, in order to minimize the time spent in JIT compilation, the scope of such useful analyses is quite limited, thereby restricting their precision significantly. On the contrary, even though it is feasible to perform precise program analyses statically, it is not possible to use their results in a managed runtime without a closed-world assumption. In this paper, we propose a static+dynamic scheme that allows one to harness the results of a precise static escape analysis for allocating objects on stack, while taking care of both soundness and efficiency concerns in the runtime.

Paper Link

Thank You!! Questions?