Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

Computer Systems in India Talk Series (Systems@India)

Aditya Anand
Advisor: Prof. Manas Thakur
Indian Institute of Technology Bombay

14th August 2024

Content of the slides

Aditya Anand[*], Solai Adithya[†], Swapnil Rustagi[†], Priyam Seth[†], Vijay Sundaresan[#], Daryl Maier[#], V Krishna Nandivada[+] and Manas Thakur[*]. “**Optimistic Stack Allocation and Dynamic Heapification in Managed Runtimes**”, ***PLDI* 2024.**

[*]IIT Bombay, [†]IIT Mandi, [#]IBM Canada, [+]IIT Madras

PLDI

COPENHAGEN 2024

Compilation in Programming Languages

Compilation in Programming Languages

- Languages like C, C++ :

Compilation in Programming Languages

- Languages like C, C++ :
  - Use static compilers (gcc, g++).
  - Generate executable which can be directly executed on machine.
  - Optimizations performed will be based on statically available information.

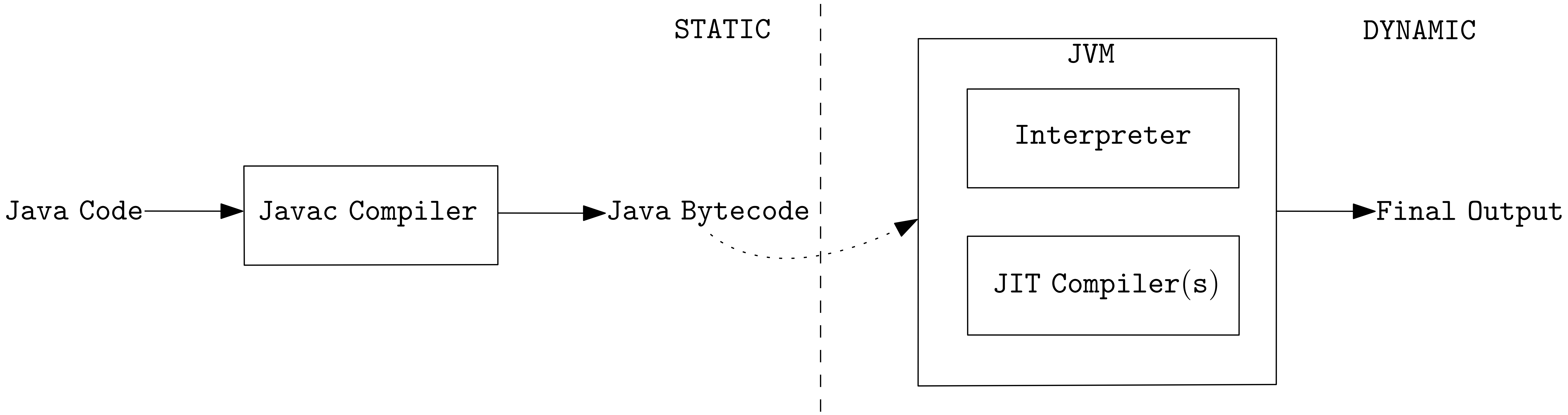
Compilation in Programming Languages

- Languages like C, C++ :
  - Use static compilers (gcc, g++).
  - Generate executable which can be directly executed on machine.
  - Optimizations performed will be based on statically available information.
- Languages like Java, C# and Scala:

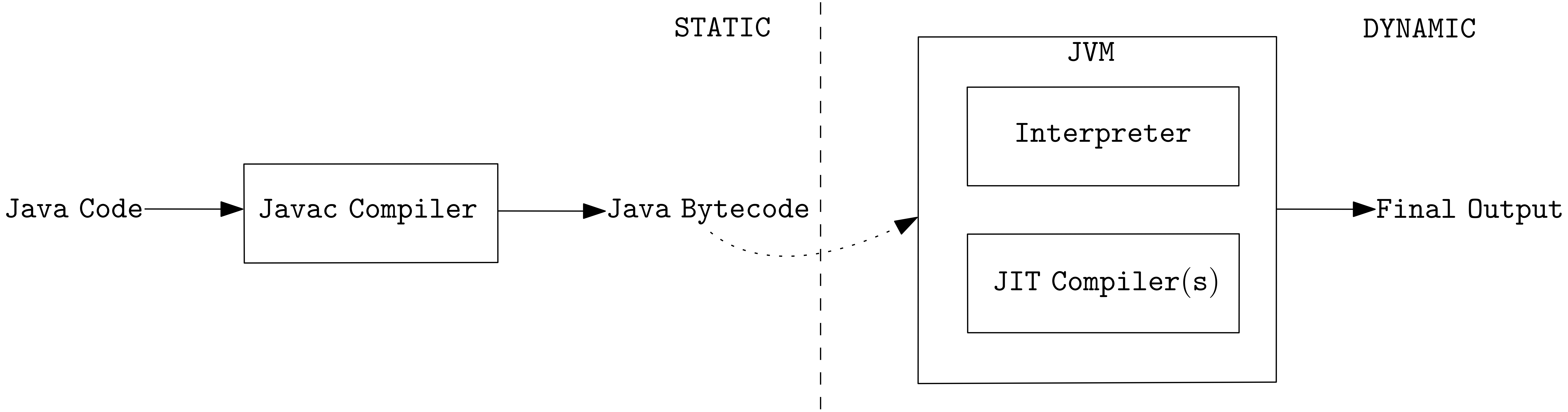
Compilation in Programming Languages

- Languages like C, C++ :
  - Use static compilers (gcc, g++).
  - Generate executable which can be directly executed on machine.
  - Optimizations performed will be based on statically available information.
- Languages like Java, C# and Scala:
  - First get compiled by a static compiler.
  - Compiled output is passed to a managed runtime for further execution.

Program Translation in Java

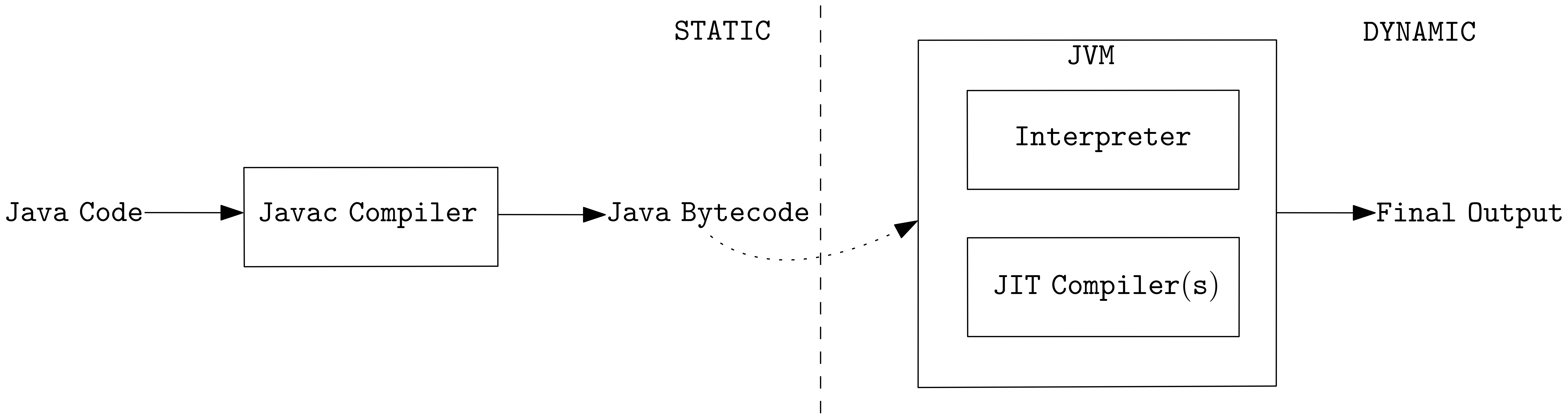


Program Translation in Java



- Static: Javac generates bytecode.

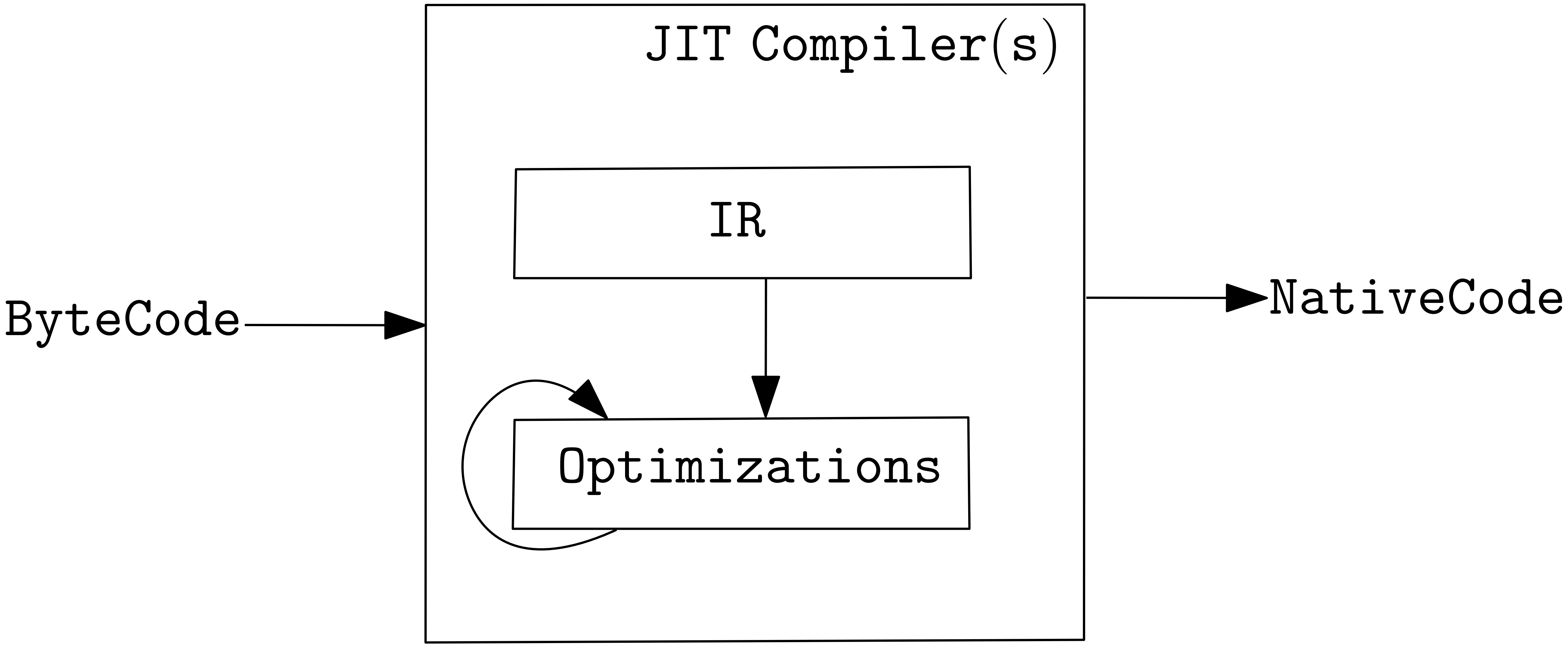
Program Translation in Java



- 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.

Objects in Java

Objects in Java

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

Objects in Java

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

Objects 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
```

Objects 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:

Objects 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.

Objects 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:

Objects 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.

Objects 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.
  - 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
    - Determines 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
    - Determines 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
    - Determines 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
  - Determines 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
    - Determines 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 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 optimistically allocate objects on stack at runtime.
- Challenges:
    - Dynamic Features: Dynamic Class Loading (DCL), Hot-Code Replacement (HCR) allows code changes.

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:
  - Dynamic Features: Dynamic Class Loading (DCL), Hot-Code Replacement (HCR) allows code changes.
  - An object that was stack allocated based on static-analysis results, might start escaping at run-time.

Static Analysis for Stack Allocation

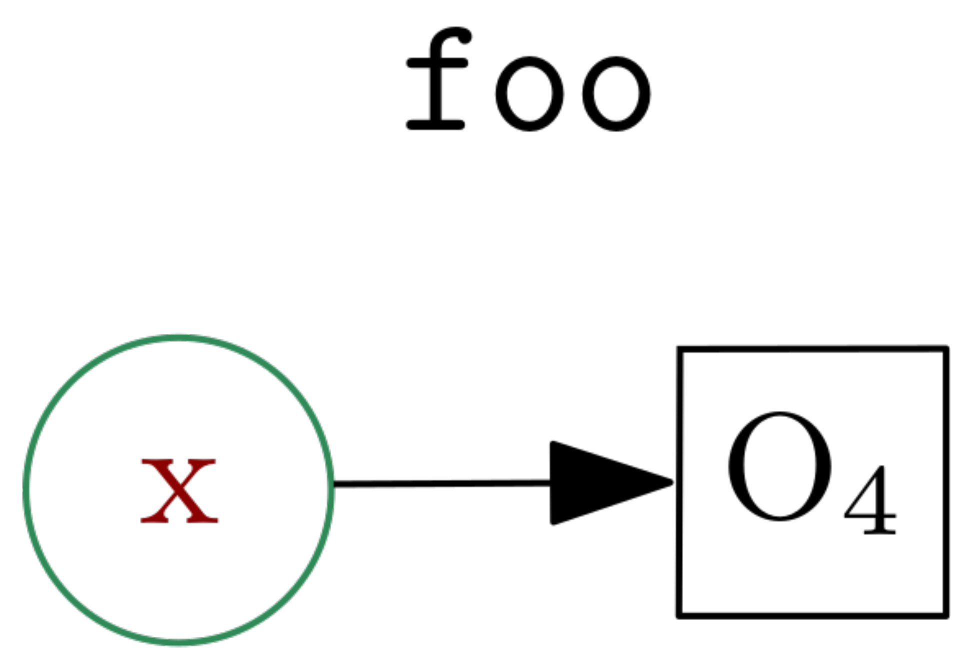
- Perform precise (context-, flow-, field-sensitive) escape analysis statically.
- Use statically generated escape analysis result to optimistically allocate objects on stack at runtime.
- Challenges:
    - Dynamic Features: Dynamic Class Loading (DCL), Hot-Code Replacement (HCR) allows code changes.
    - An object that was stack allocated based on static-analysis results, might start escaping at run-time.
- How to safely allocate objects on stack in a managed runtime?

Motivating 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 */
```

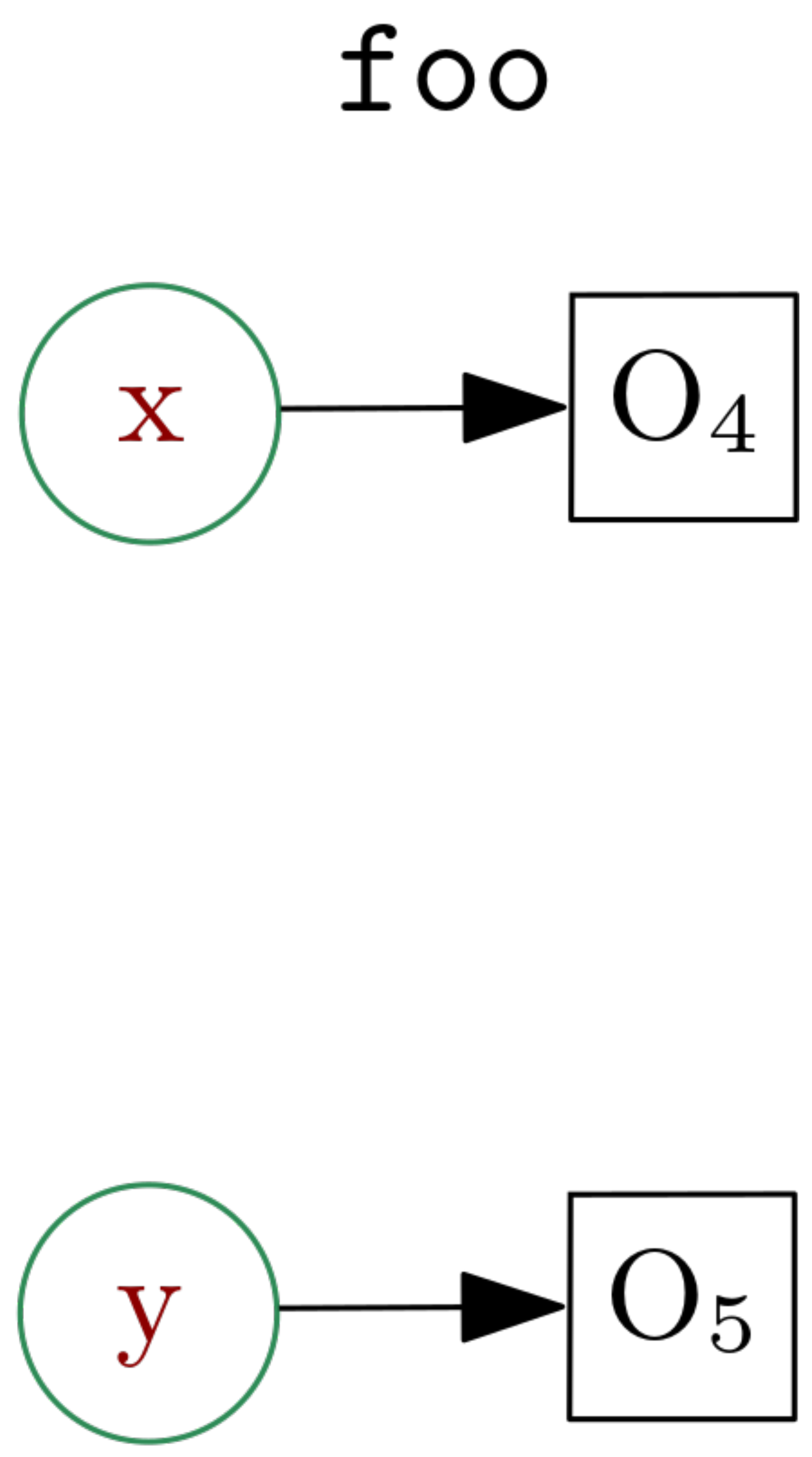
Motivating Example

```
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 */
```



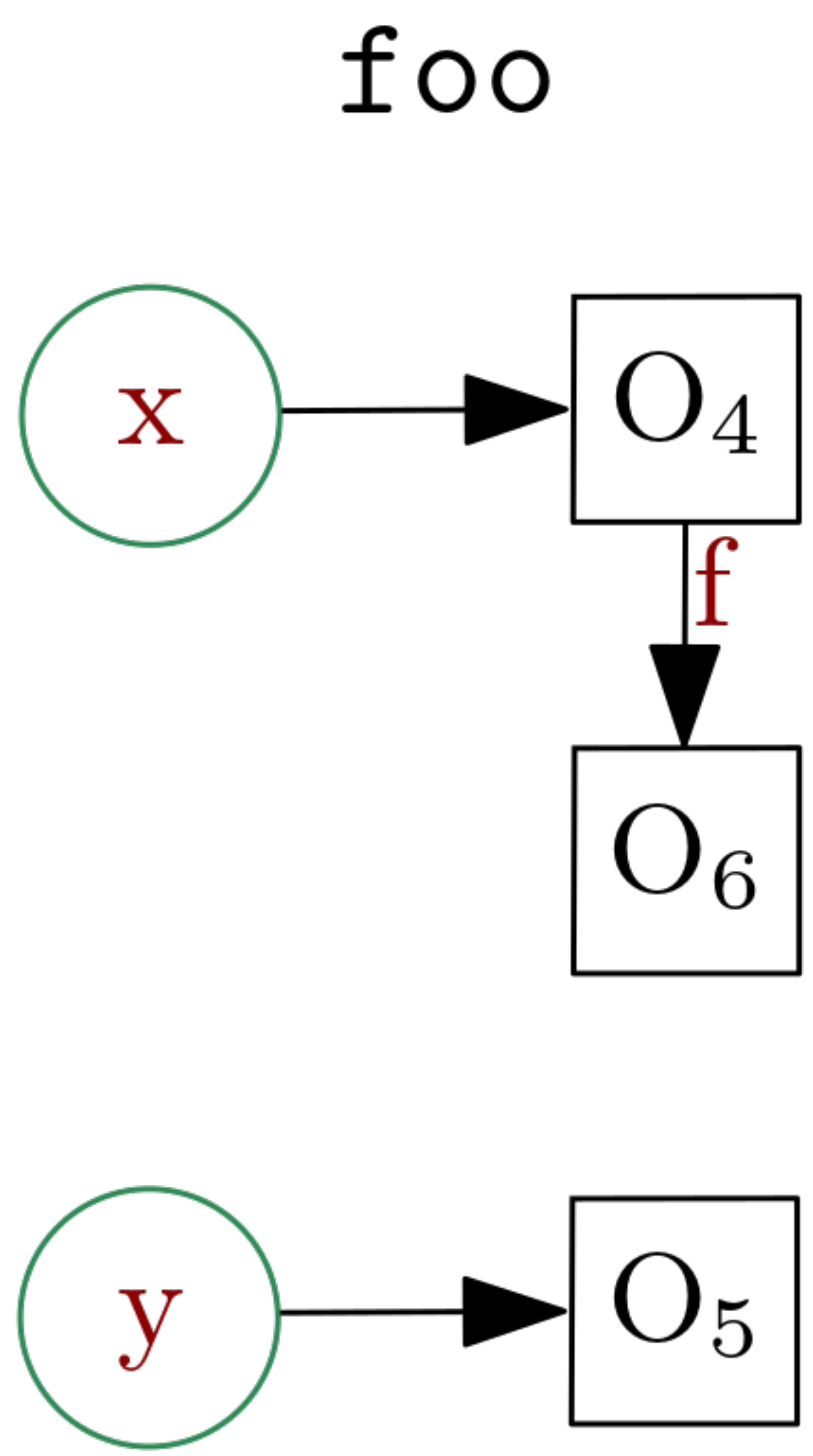
Motivating Example

```
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 */
```



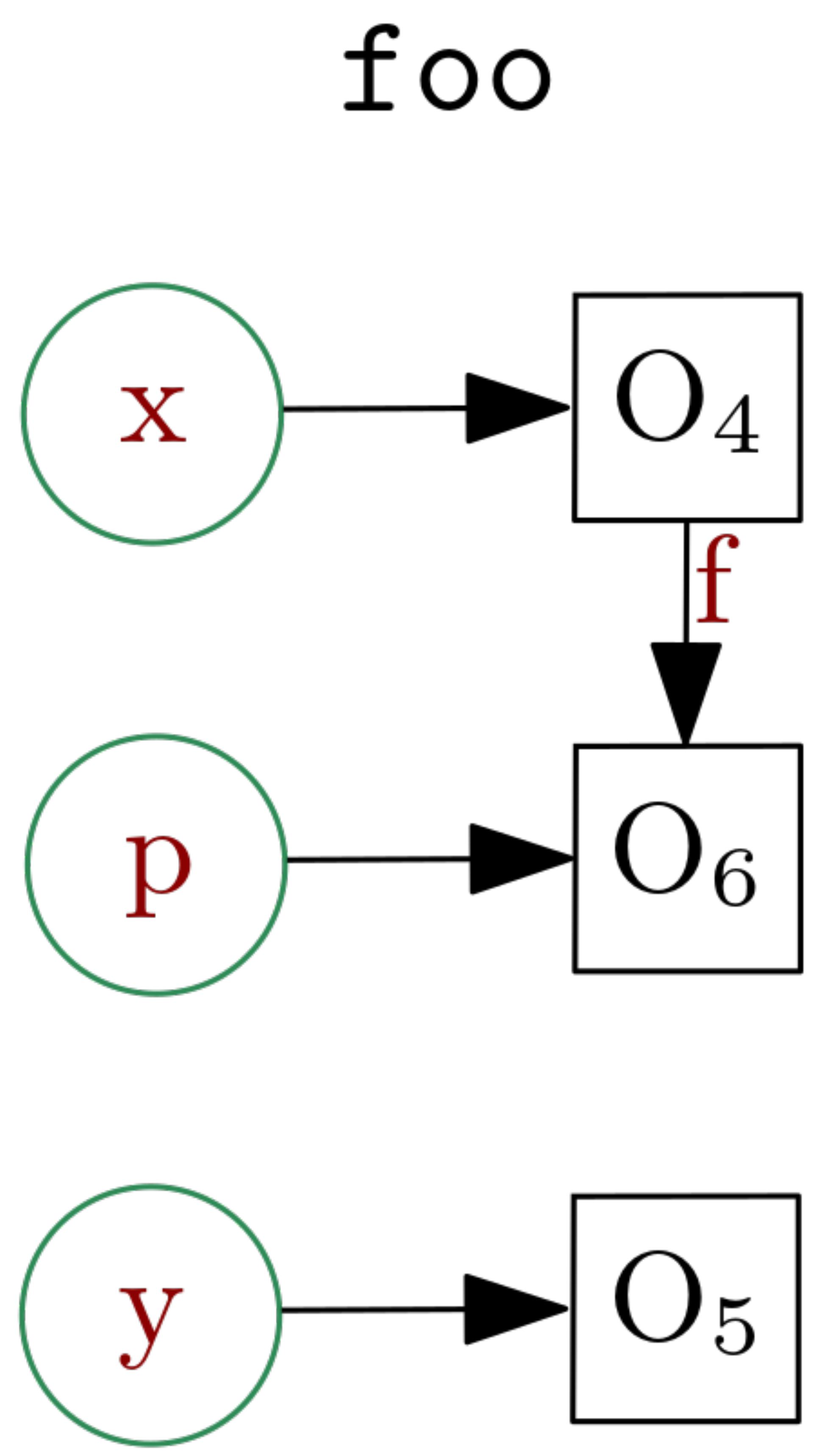
Motivating Example

```
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 */
```



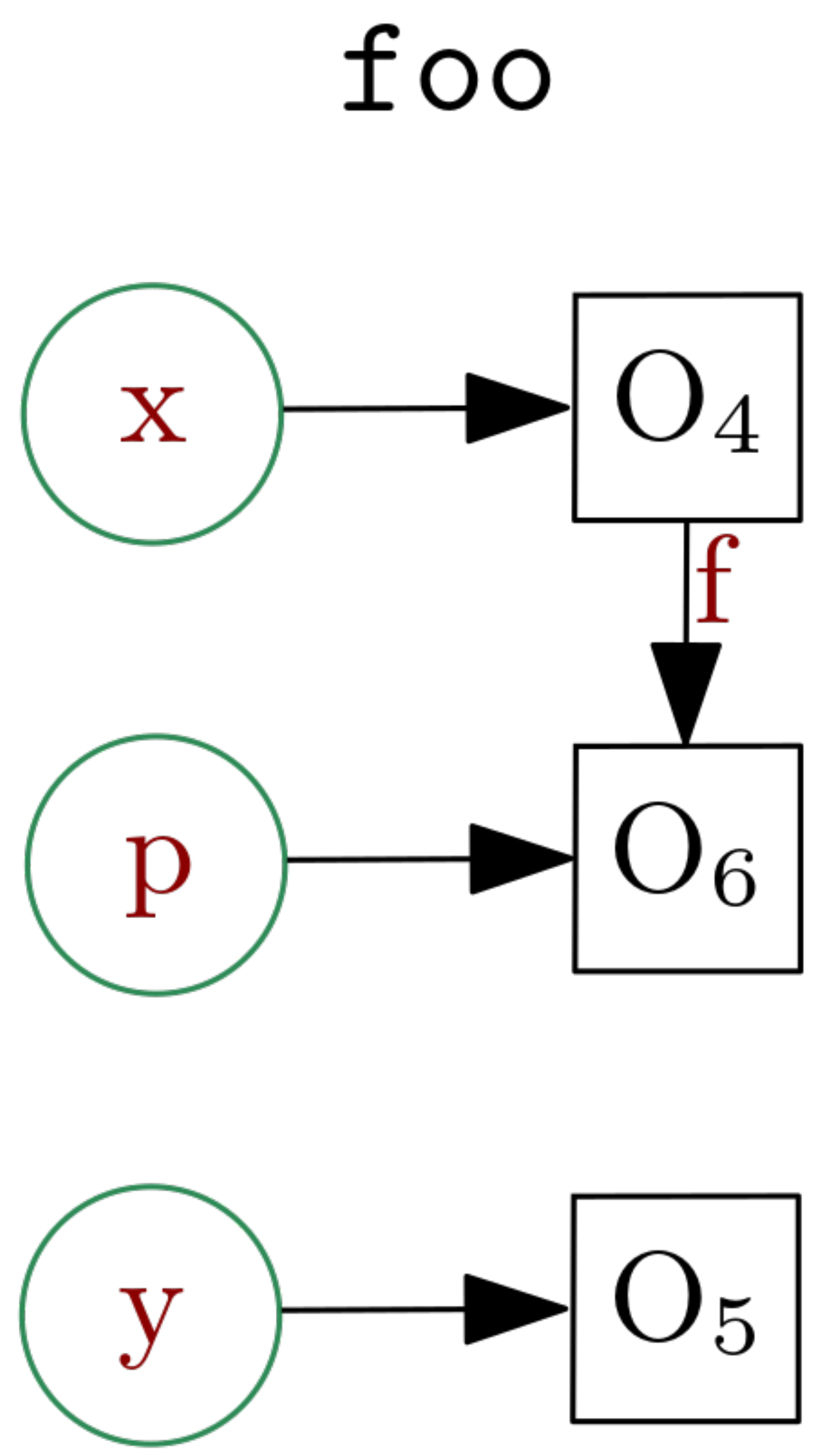
Motivating Example

```
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 */
```



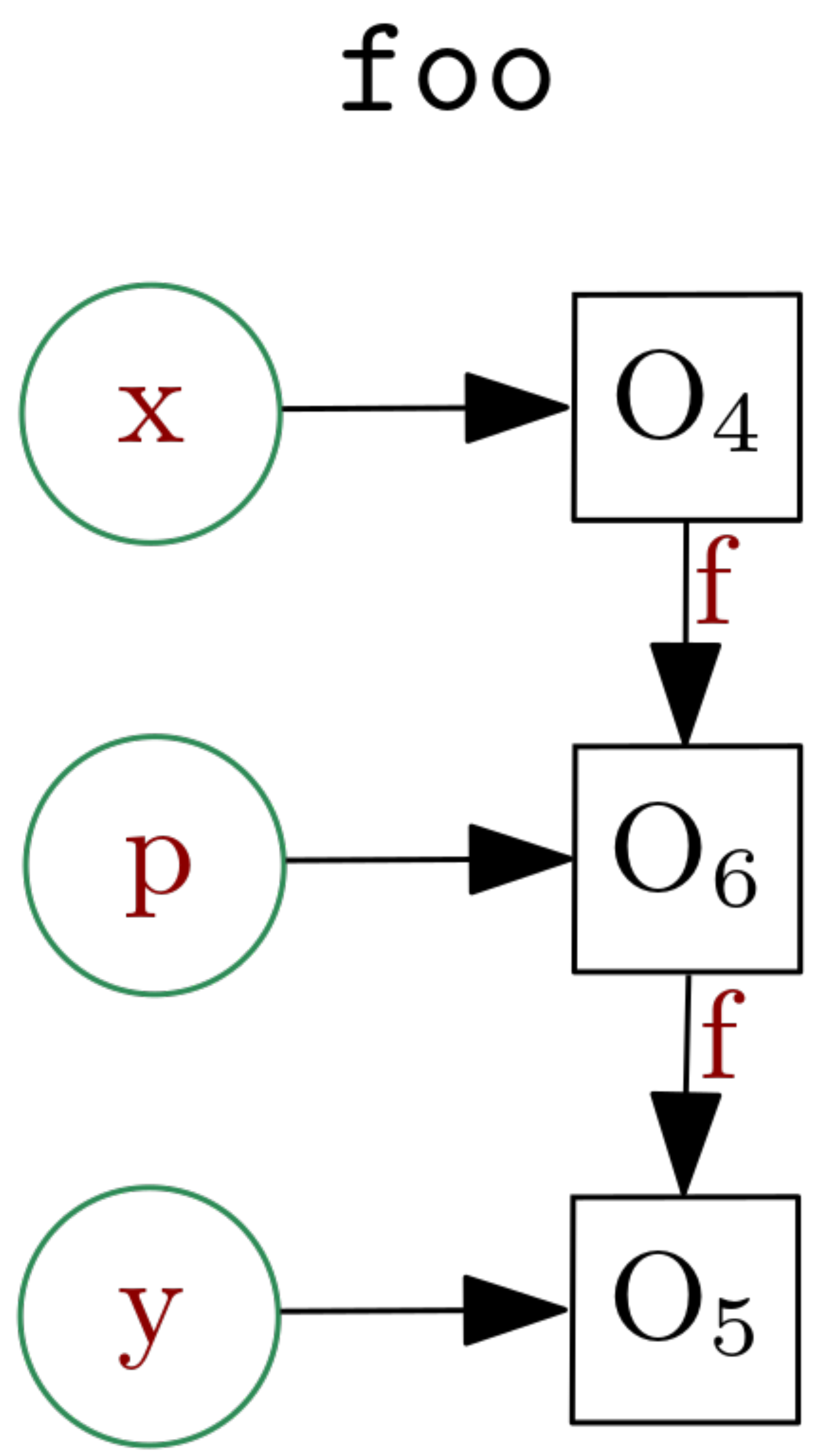
Motivating Example

```
1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.      A x = new A();   // O4
5.      A y = new A();   // O5
6.      x.f = new A();   // O6
7.      A p = x.f;
8.      bar(p, y);
9.      r.zar(p, q);
10.  } /* method foo */
11.    void zar(A p, A q) { . . .}
12.    void bar(A p1, A p2) {
13.       p1.f = p2;
14.    } /* method bar */
15. } /* class A */
```



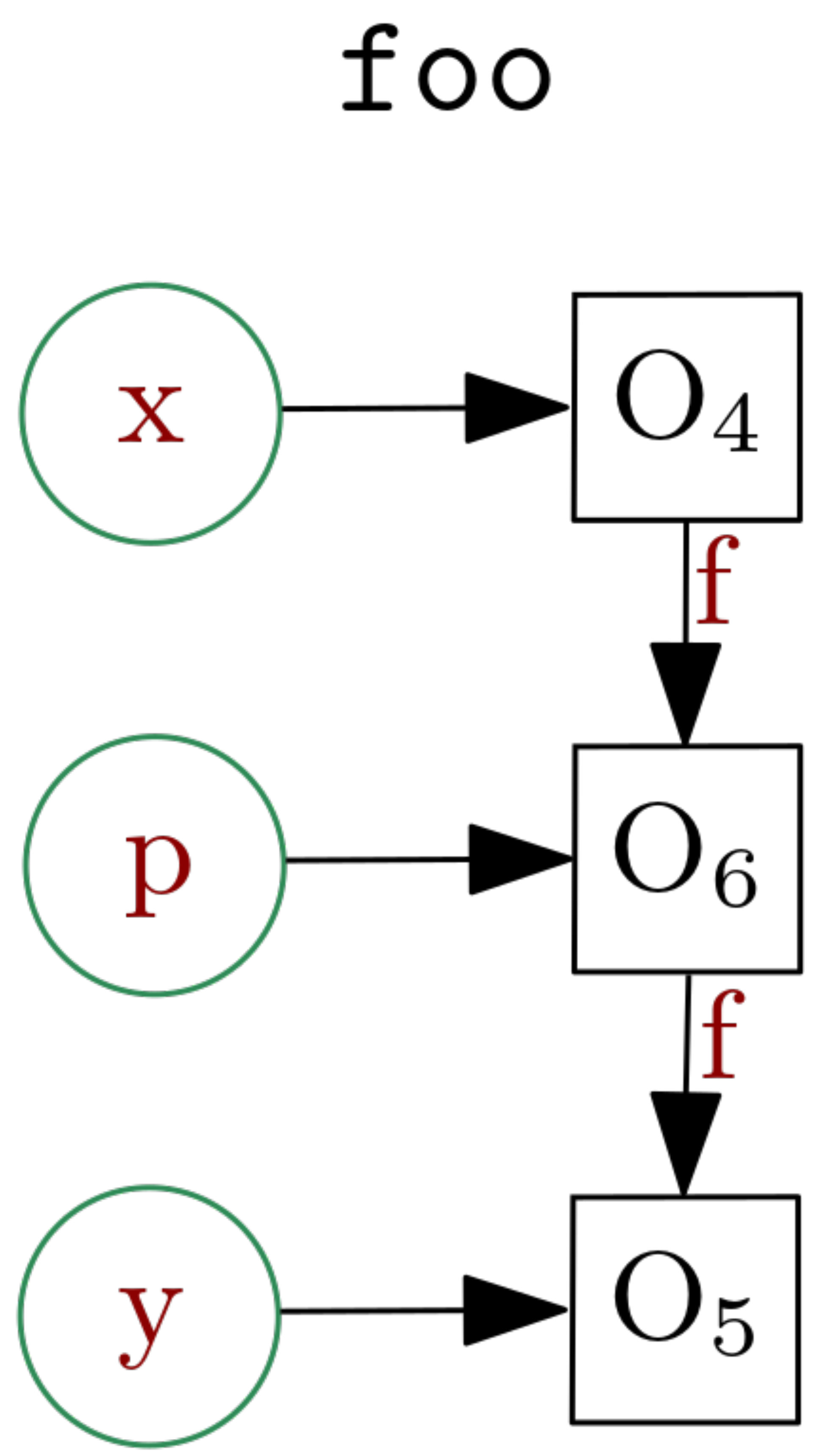
Motivating Example

```
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 */
```



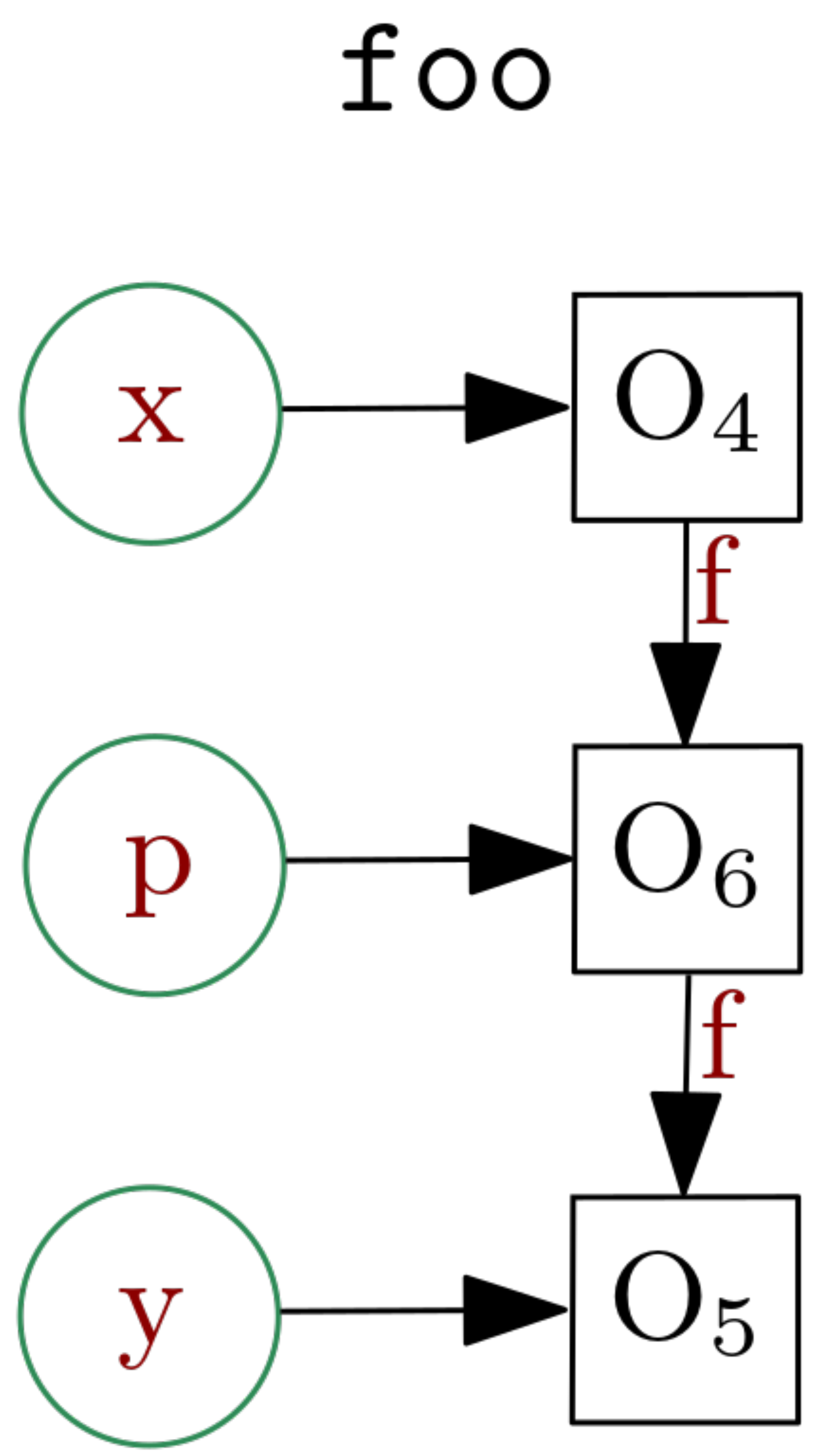
Motivating Example

```
1. class A {
2.    A f;
3.    void foo(A q, A r) {
4.       A x = new A();   // O4
5.       A y = new A();   // O5
6.       x.f = new A();   // O6
7.       A p = x.f;
8.       bar(p, y);
9.       r.zar(p, q);
10.   } /* method foo */
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.      p1.f = p2;
14.   } /* method bar */
15. } /* class A */
```



Motivating 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 */
```

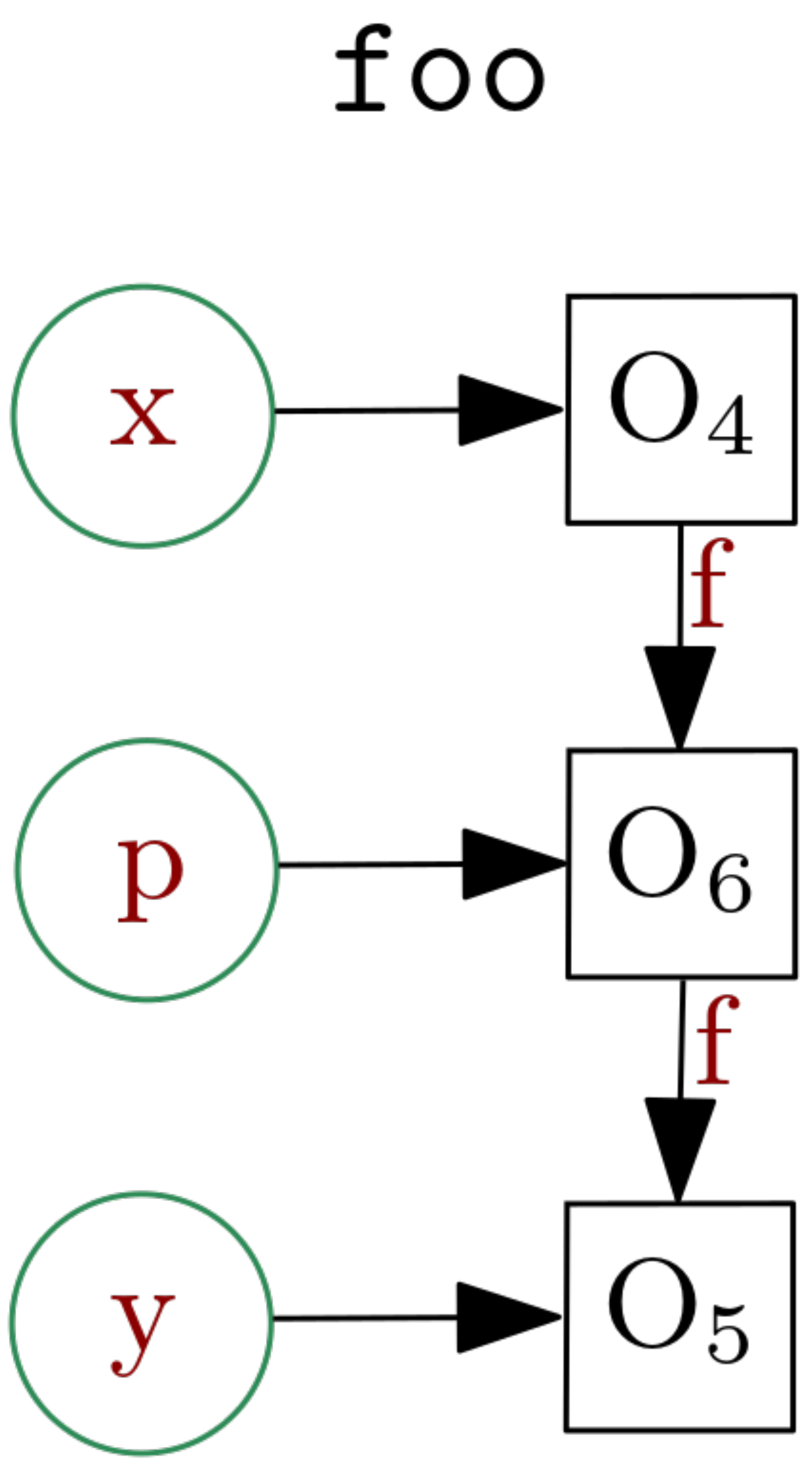


Stack Allocate

O_4, O_5 and O_6

Motivating Example

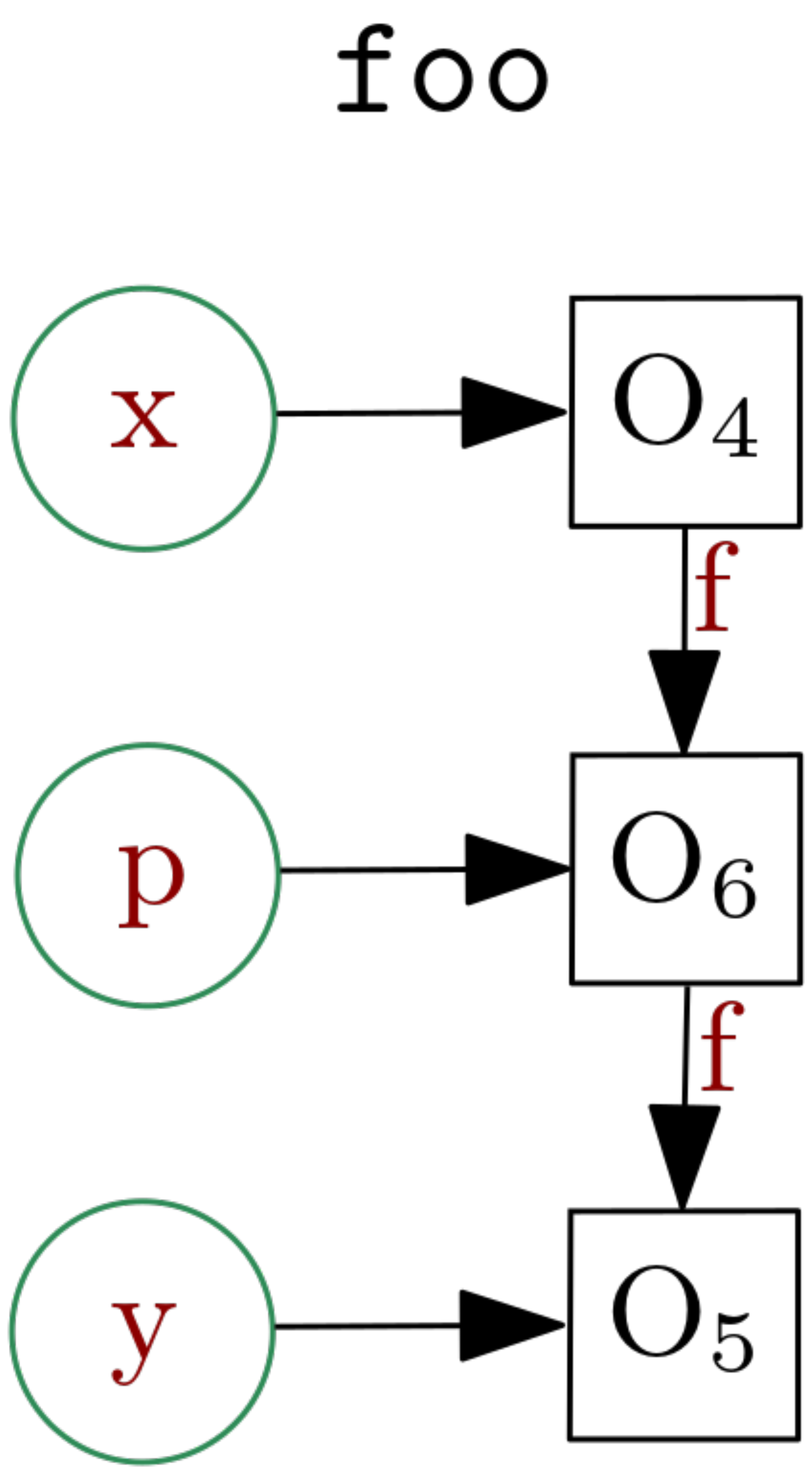
```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();   // O4
     A y = new A();   // O5
     x.f = new A();   // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class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



Motivating Example

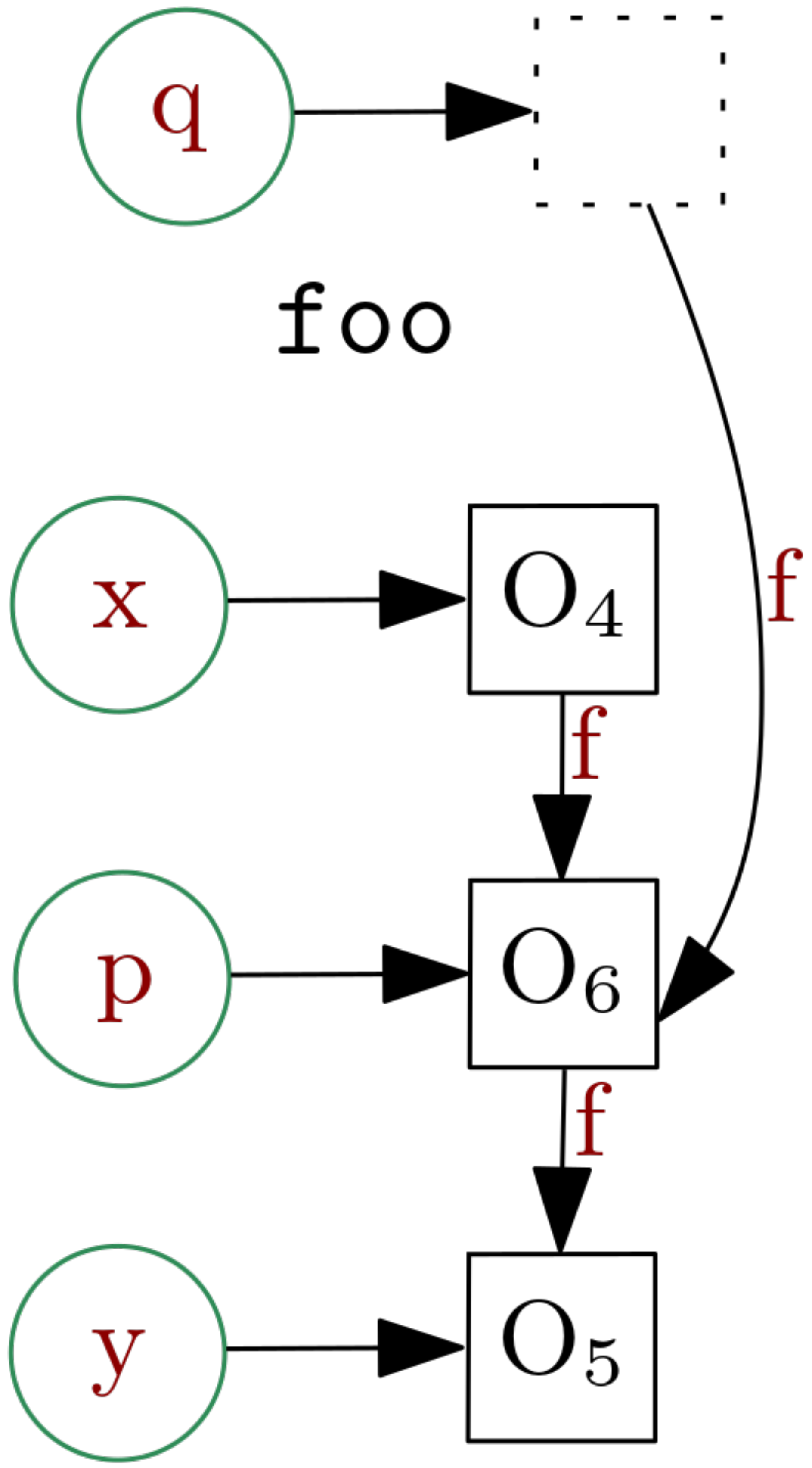
```
class A {
  A f;
  void foo(A q, A r) {
    A x = new A();   // O4
    A y = new A();   // O5
    x.f = new A();   // O6
    A p = x.f;
    bar(p, y);
    r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
    p1.f = p2;
  } /* method bar */
} /* class A */
class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

Dynamically loaded



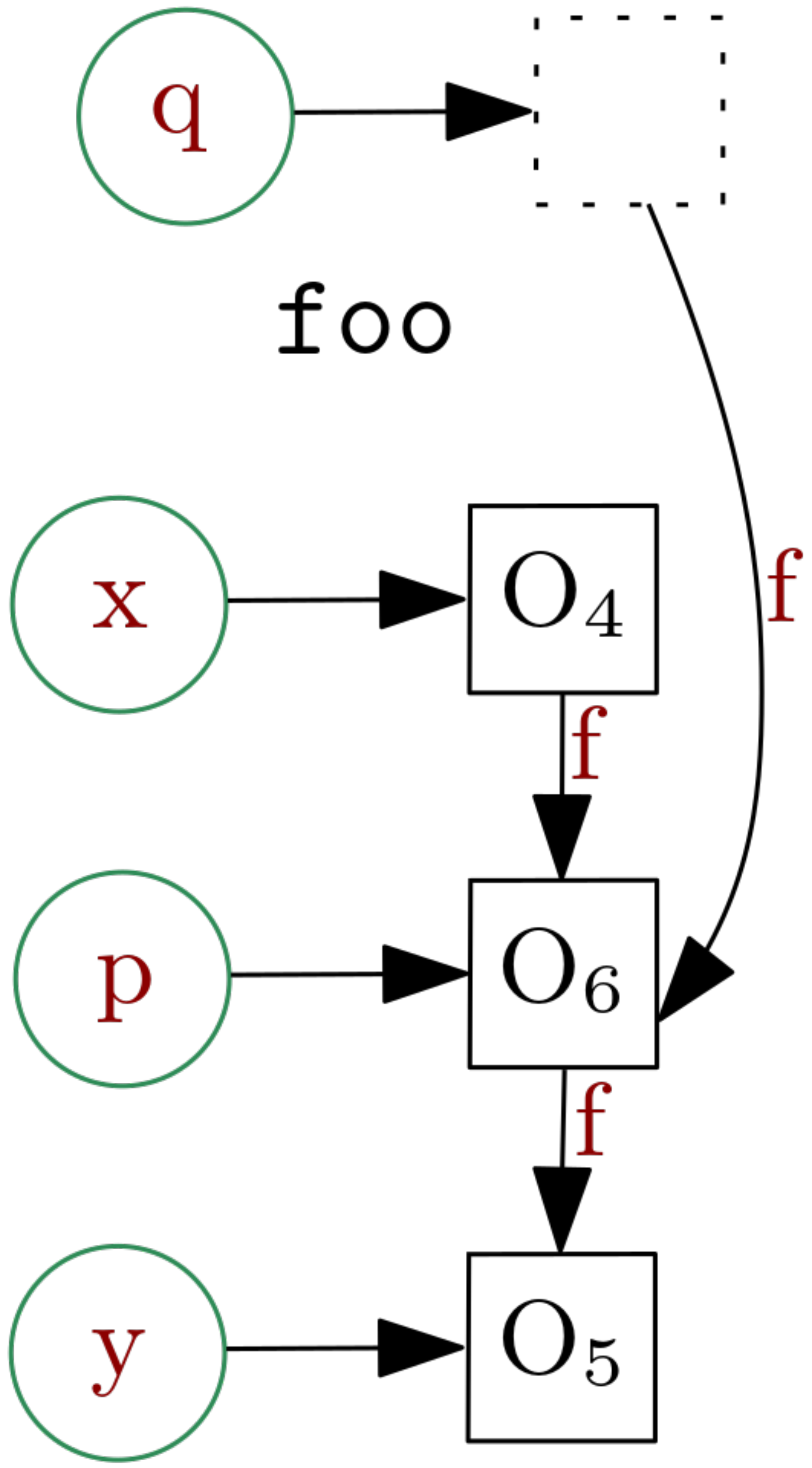
Motivating Example

```
1. class A {
2.   A f;
3.   void foo(A q, A r) {
4.      A x = new A();   // O4
5.      A y = new A();   // O5
6.      x.f = new A();   // O6
7.      A p = x.f;
8.      bar(p, y);
9.      r.zar(p, q);
10.  } /* method foo */
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.      p1.f = p2;
14.   } /* method bar */
15. } /* class A */
16. class B extends A
17.   void zar(A p, A q) {
18.      q.f = p;
19.   } /* method zar */
20. } /* class B */
```



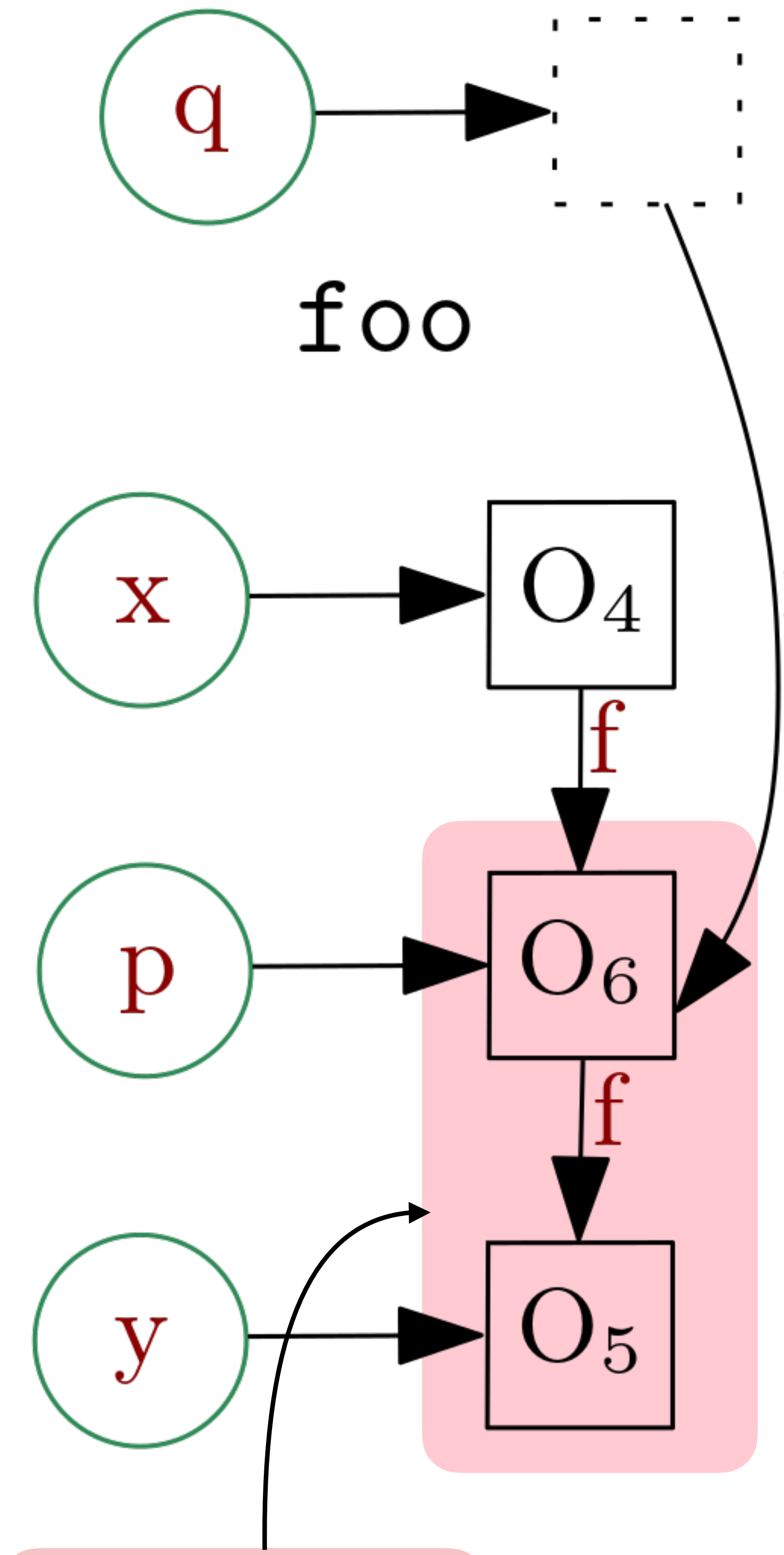
Motivating Example

```
1.  class A {
2.    A f;
3.    void foo(A q, A r) {
4.      A x = new A();   // O4
5.      A y = new A();   // O5
6.      x.f = new A();   // O6
7.      A p = x.f;
8.      bar(p, y);
9.      r.zar(p, q);
10.   } /* method foo */
11.   void zar(A p, A q) { . . .}
12.   void bar(A p1, A p2) {
13.     p1.f = p2;
14.   } /* method bar */
15. } /* class A */
16. class B extends A
17.   void zar(A p, A q) {
18.     q.f = p;
19.   } /* method zar */
20. } /* class B */
```



Motivating Example

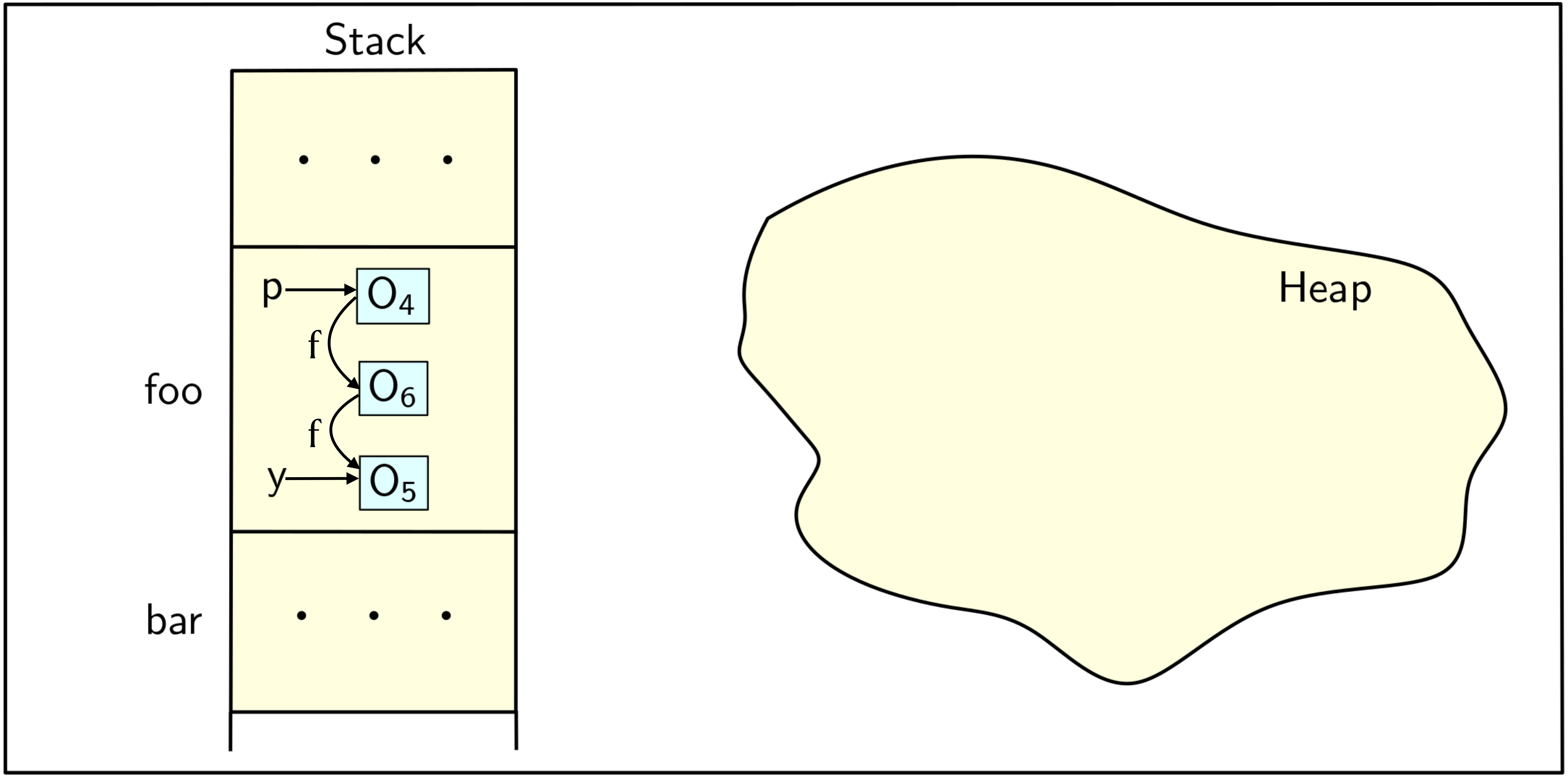
```
class A {
  A f;
  void foo(A q, A r) {
     A x = new A();  // O4
     A y = new A();  // O5
     x.f = new A();  // O6
     A p = x.f;
     bar(p, y);
     r.zar(p, q);
  } /* method foo */
  void zar(A p, A q) { . . .}
  void bar(A p1, A p2) {
     p1.f = p2;
  } /* method bar */
} /* class A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



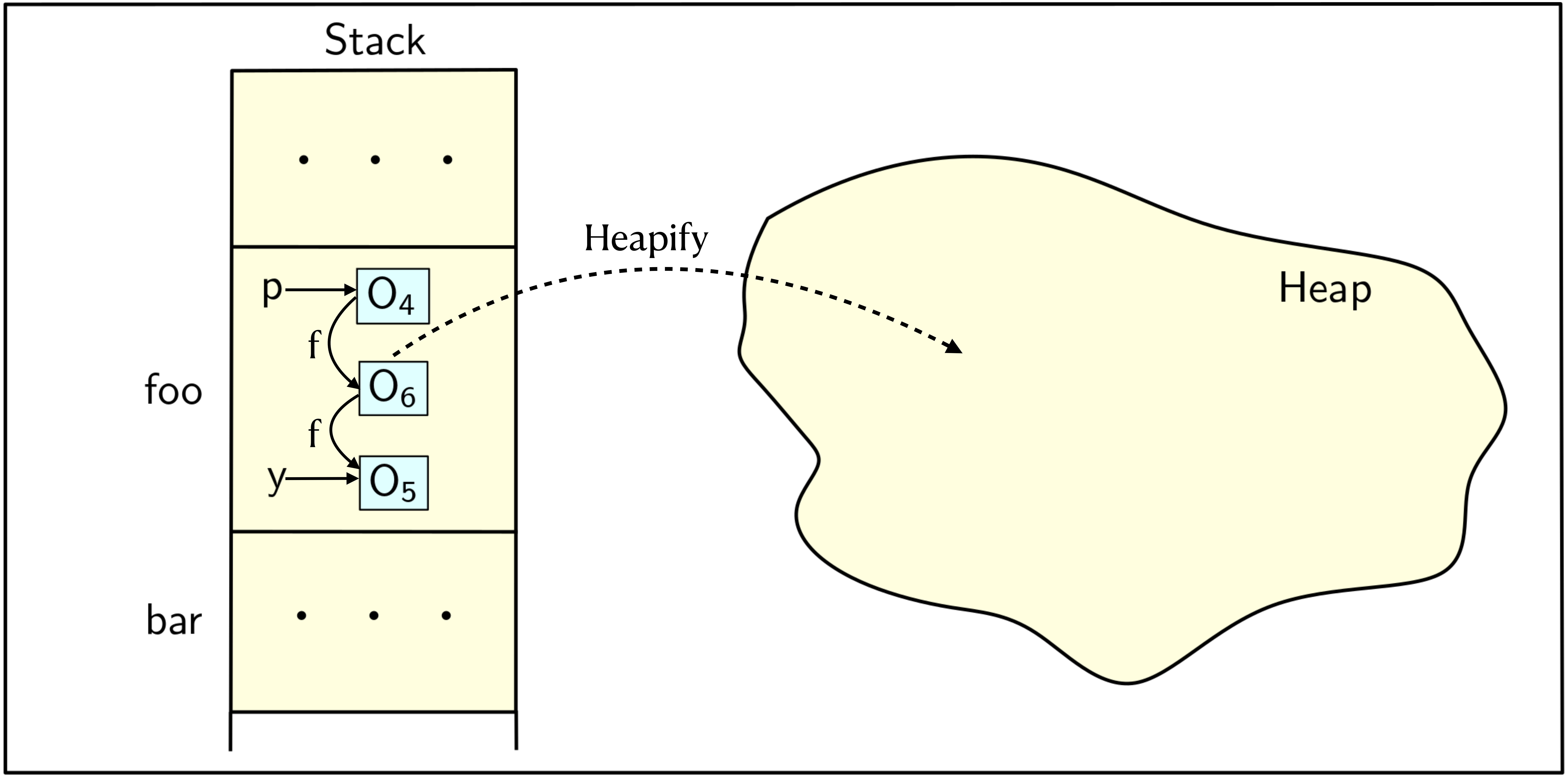
Incorrect allocation on stack

Dynamic Heapification

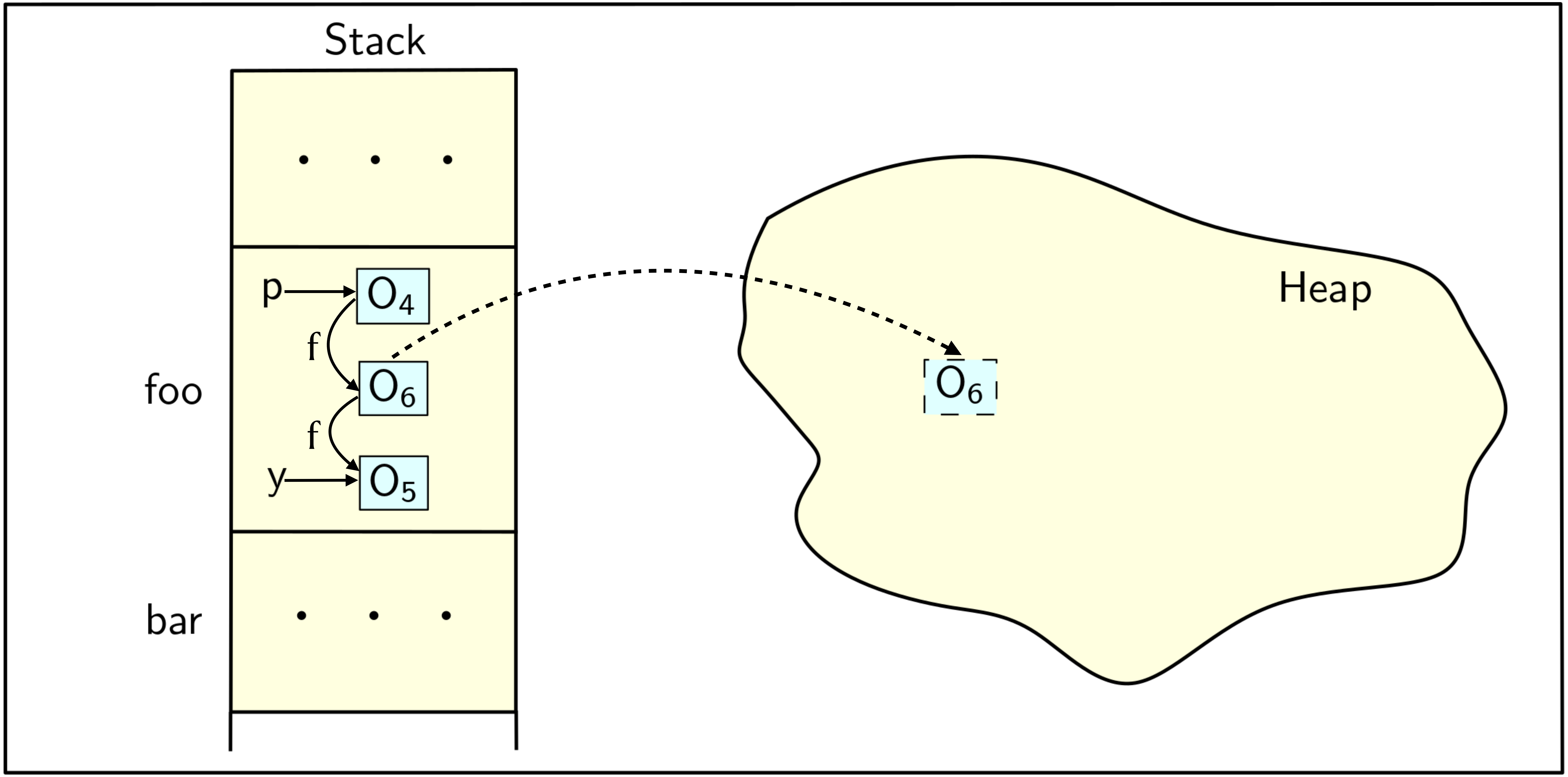
Heapification



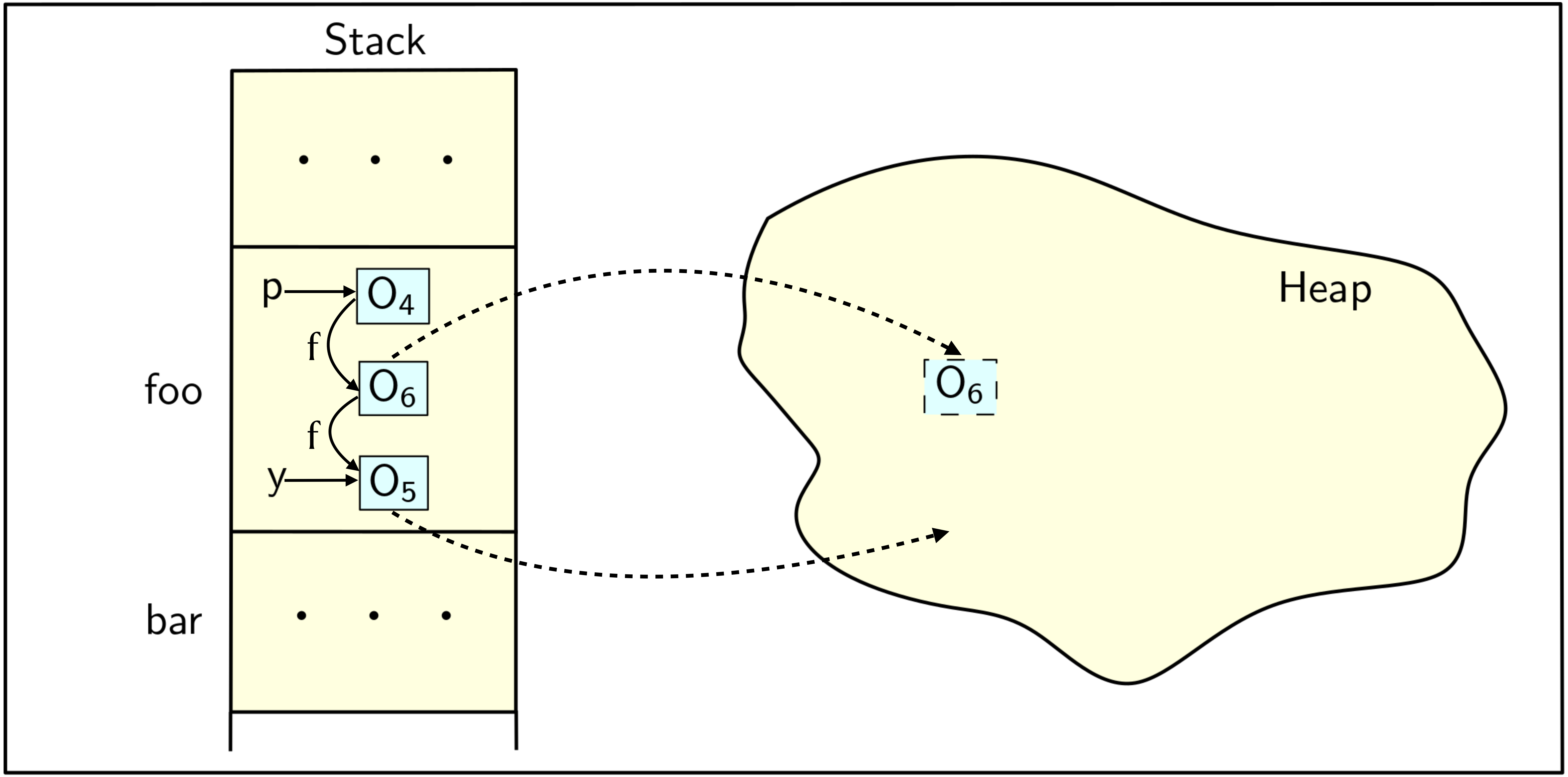
Heapification



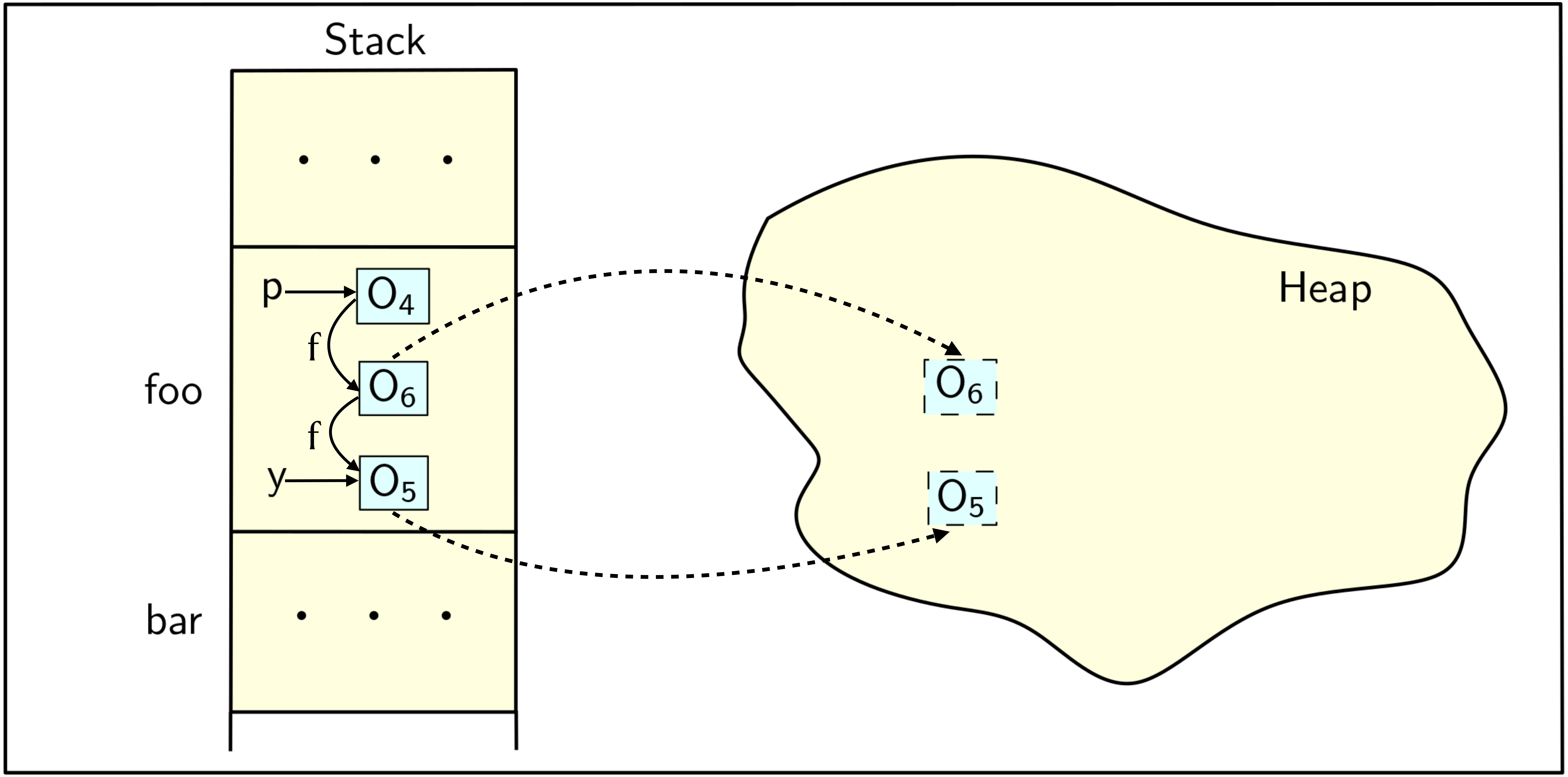
Heapification



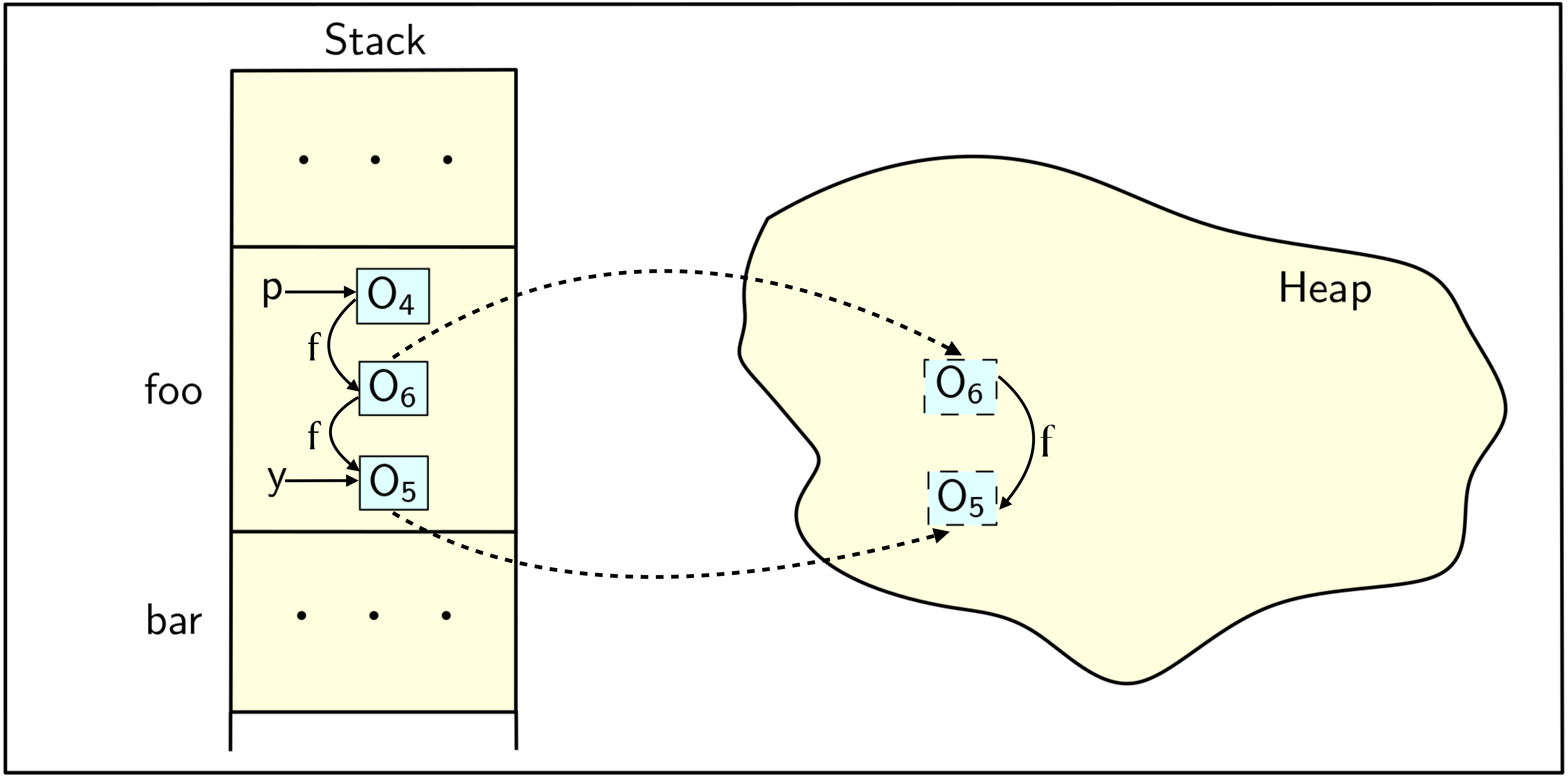
Heapification



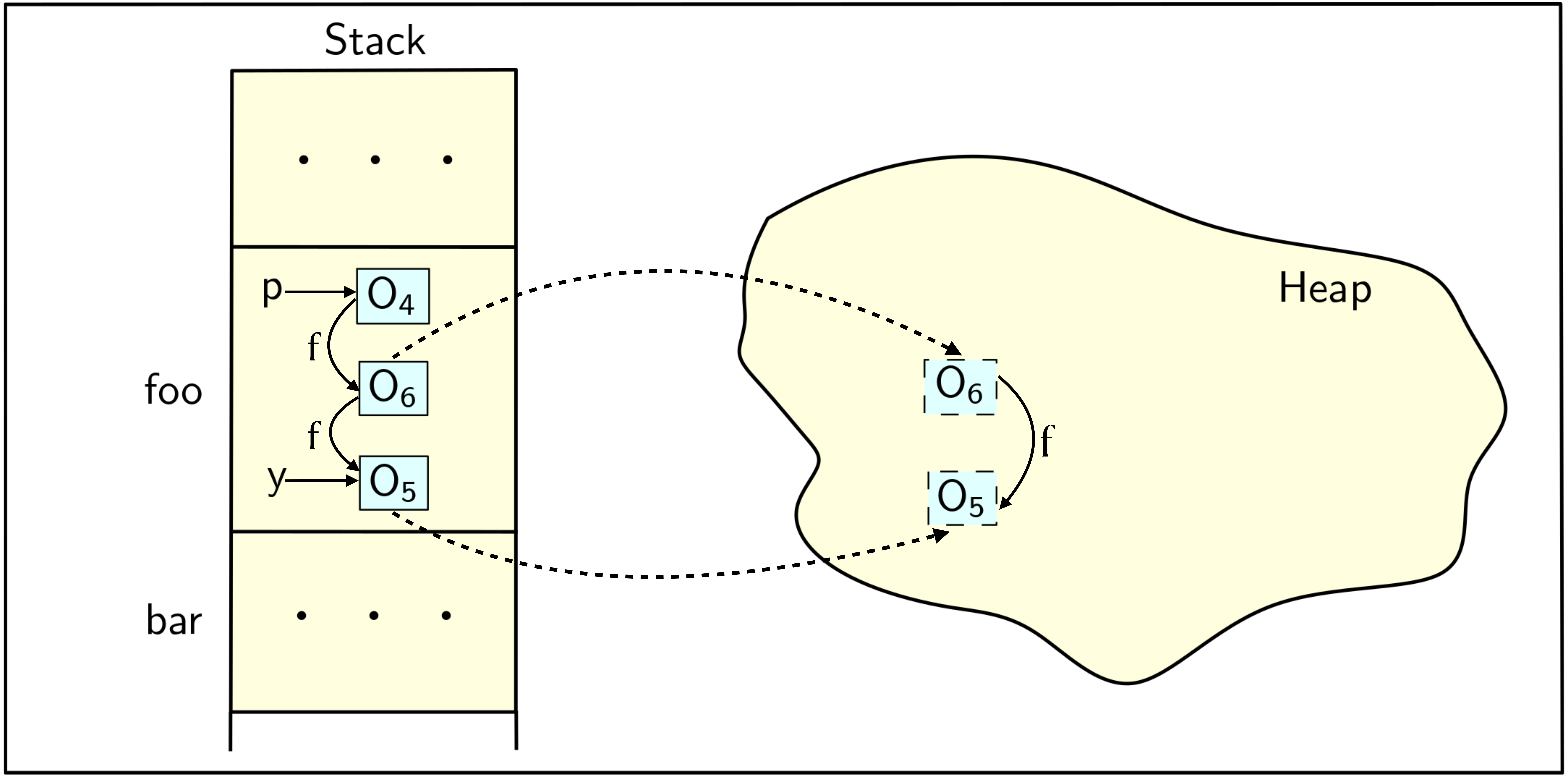
Heapification



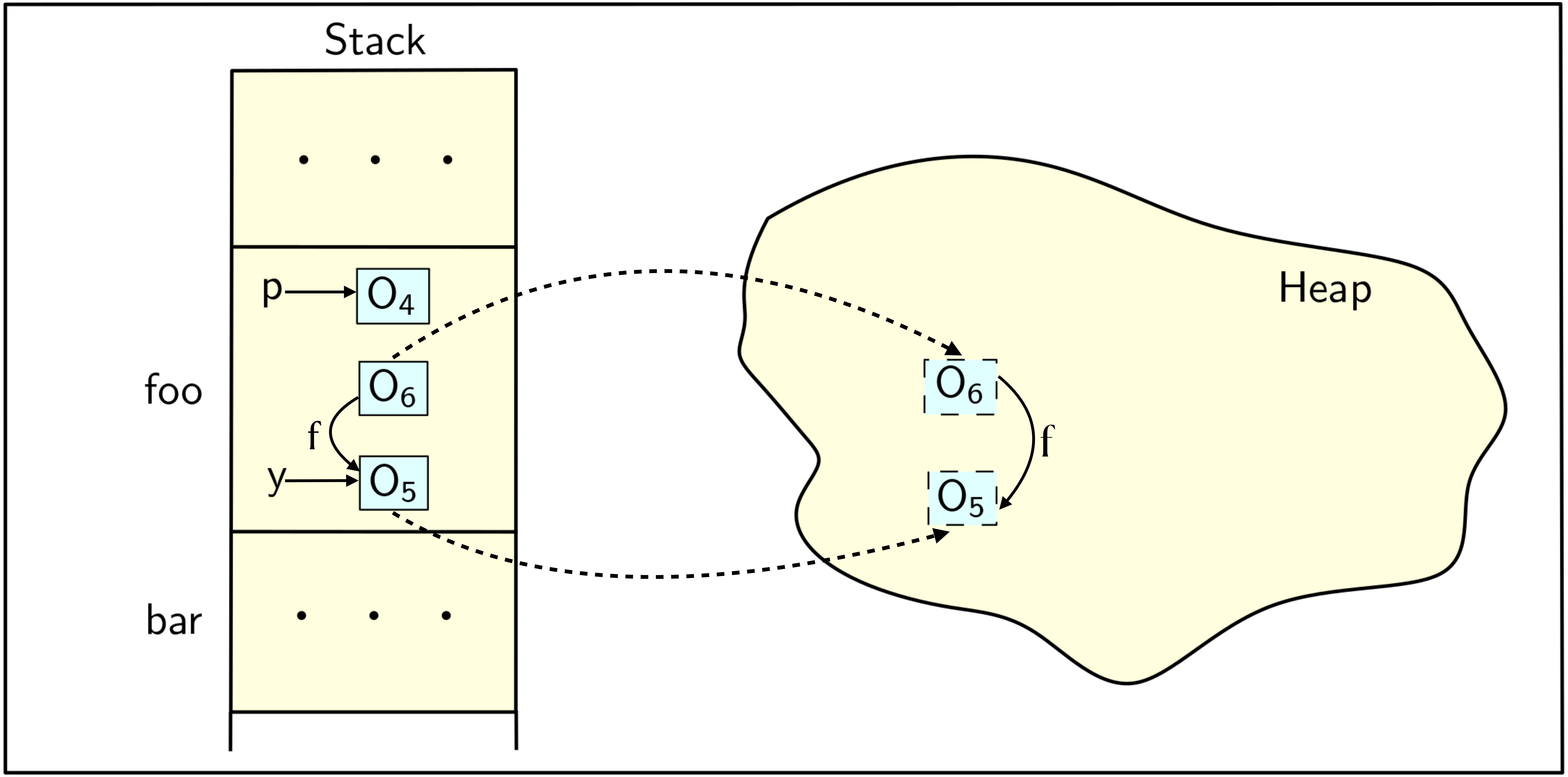
Heapification



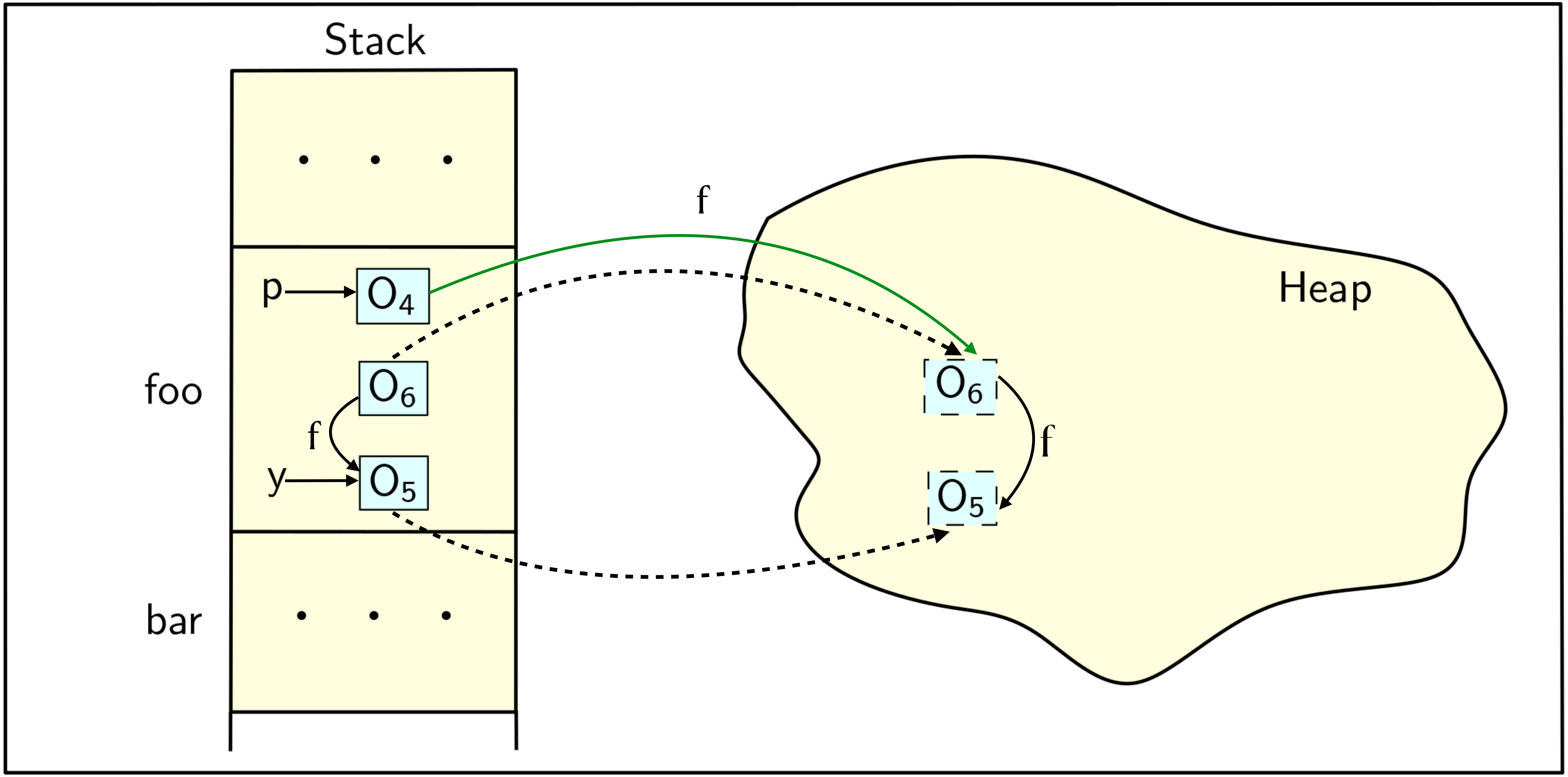
Heapification



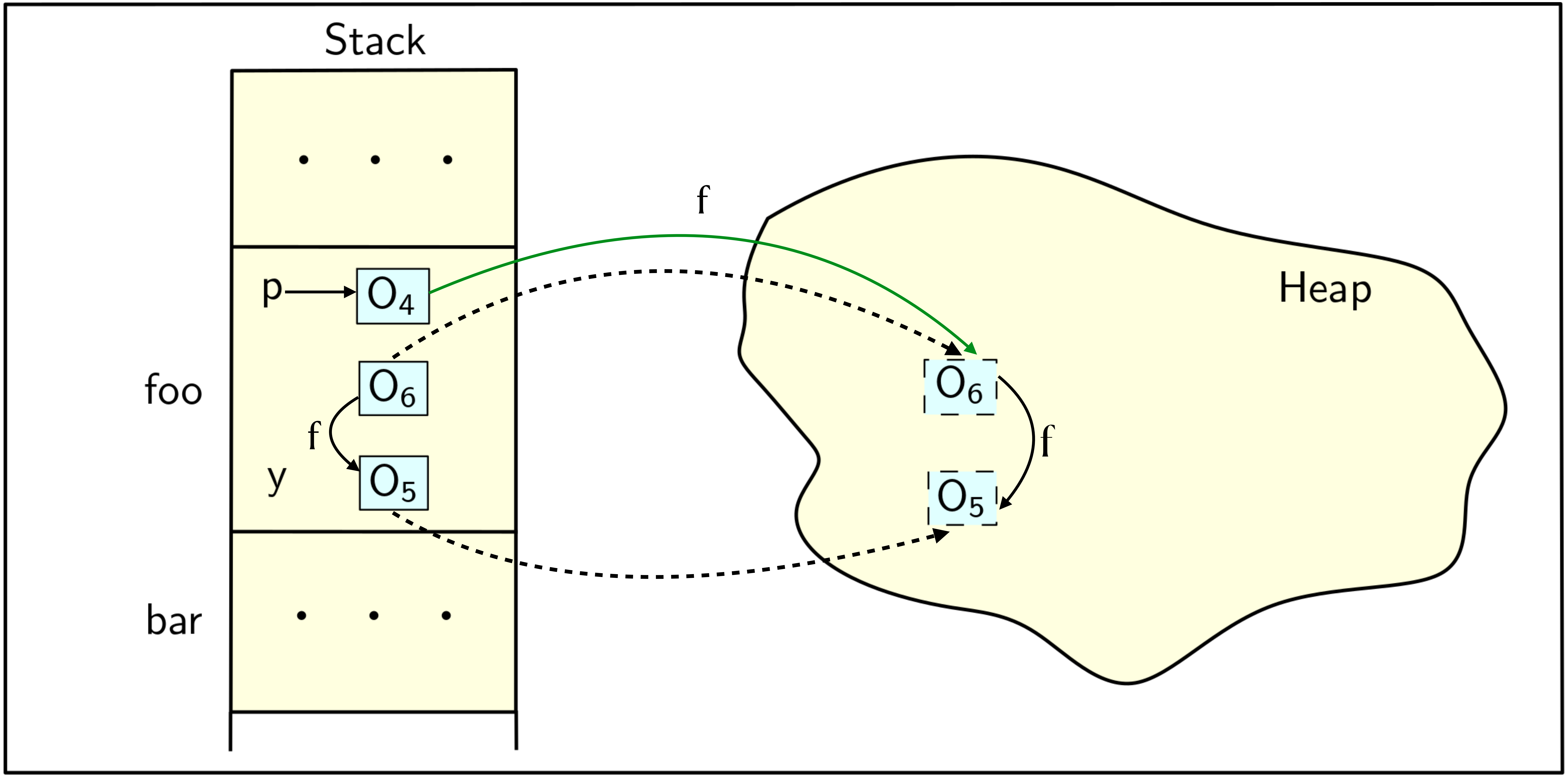
Heapification



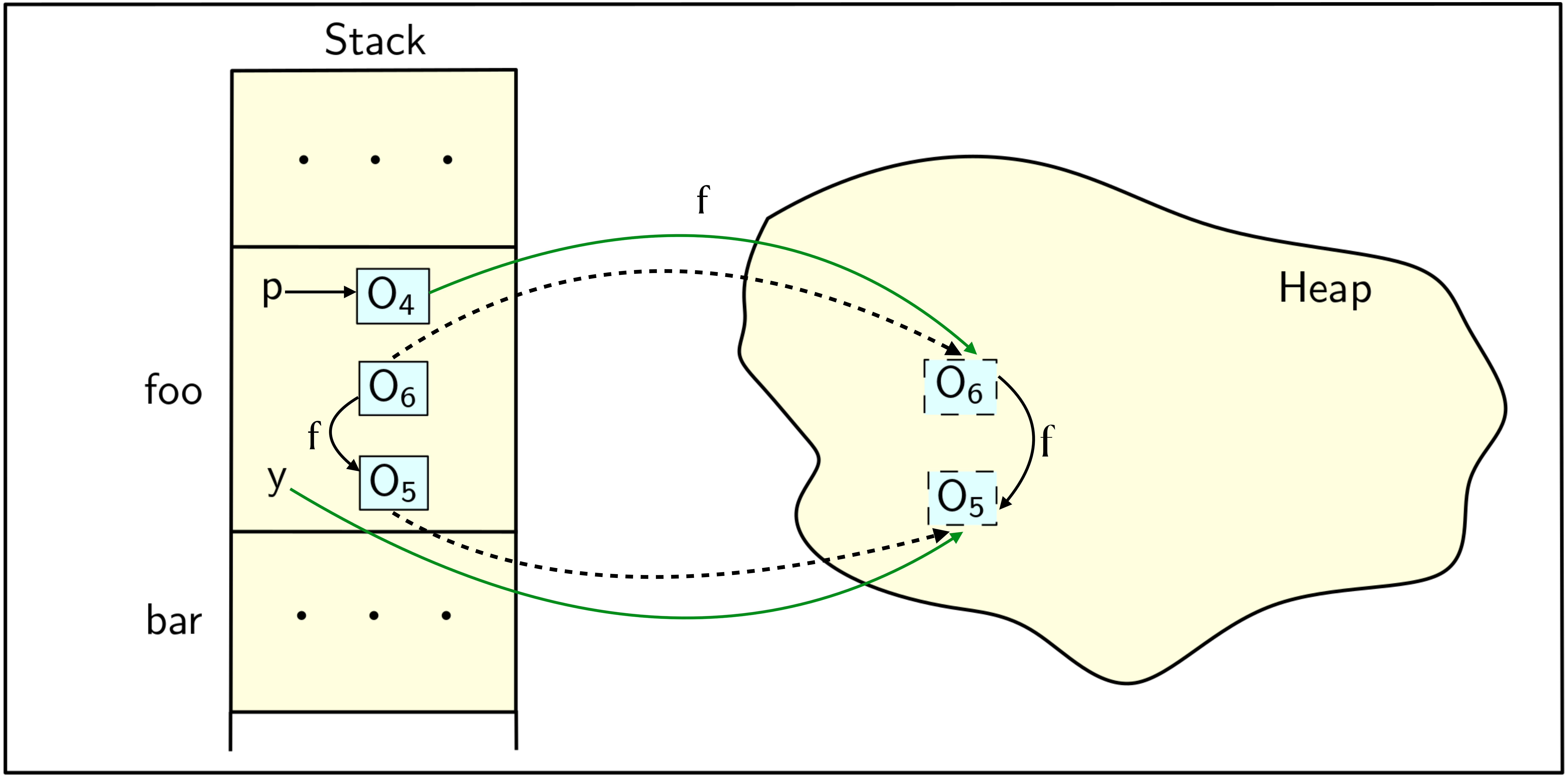
Heapification



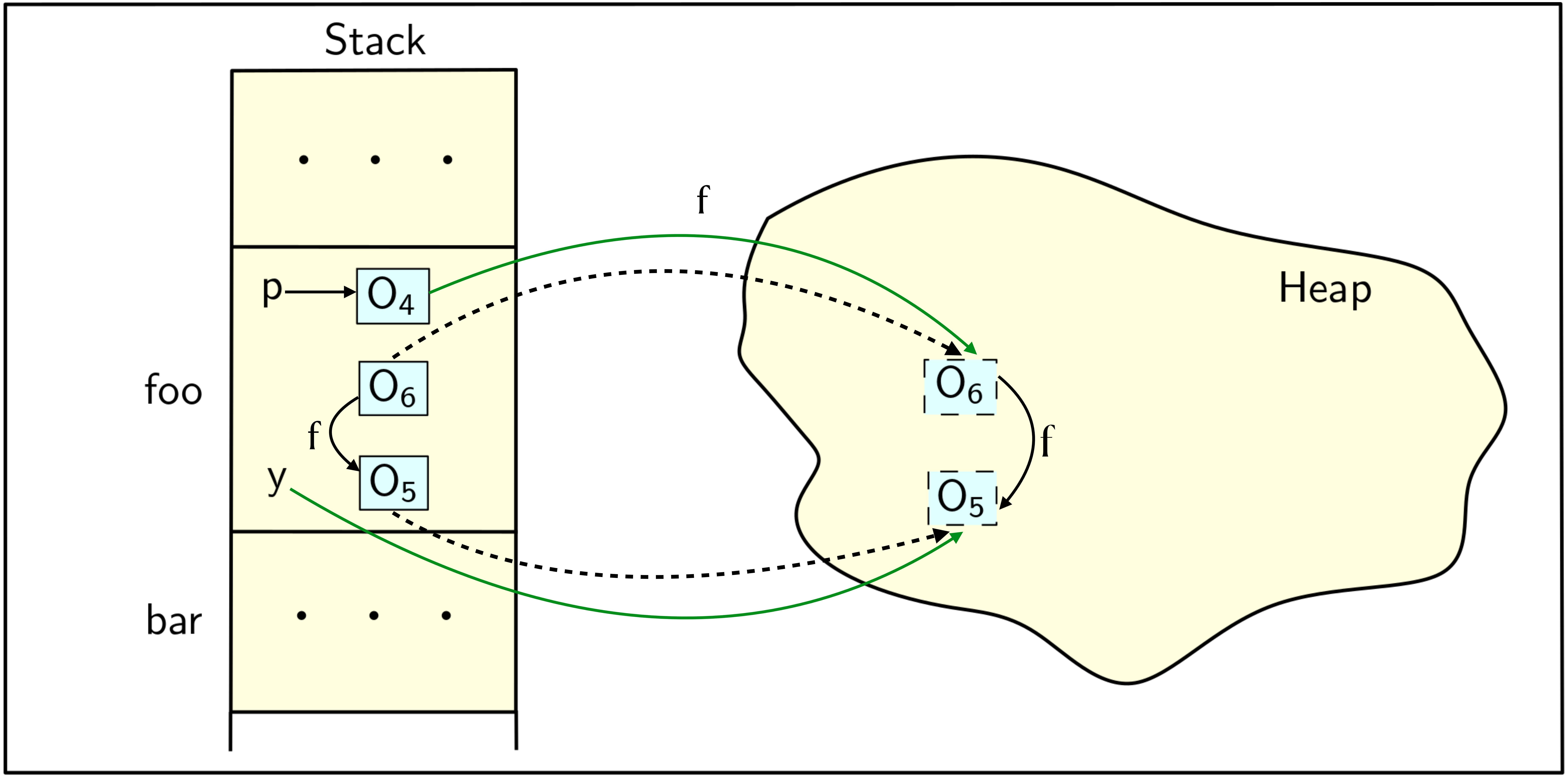
Heapification



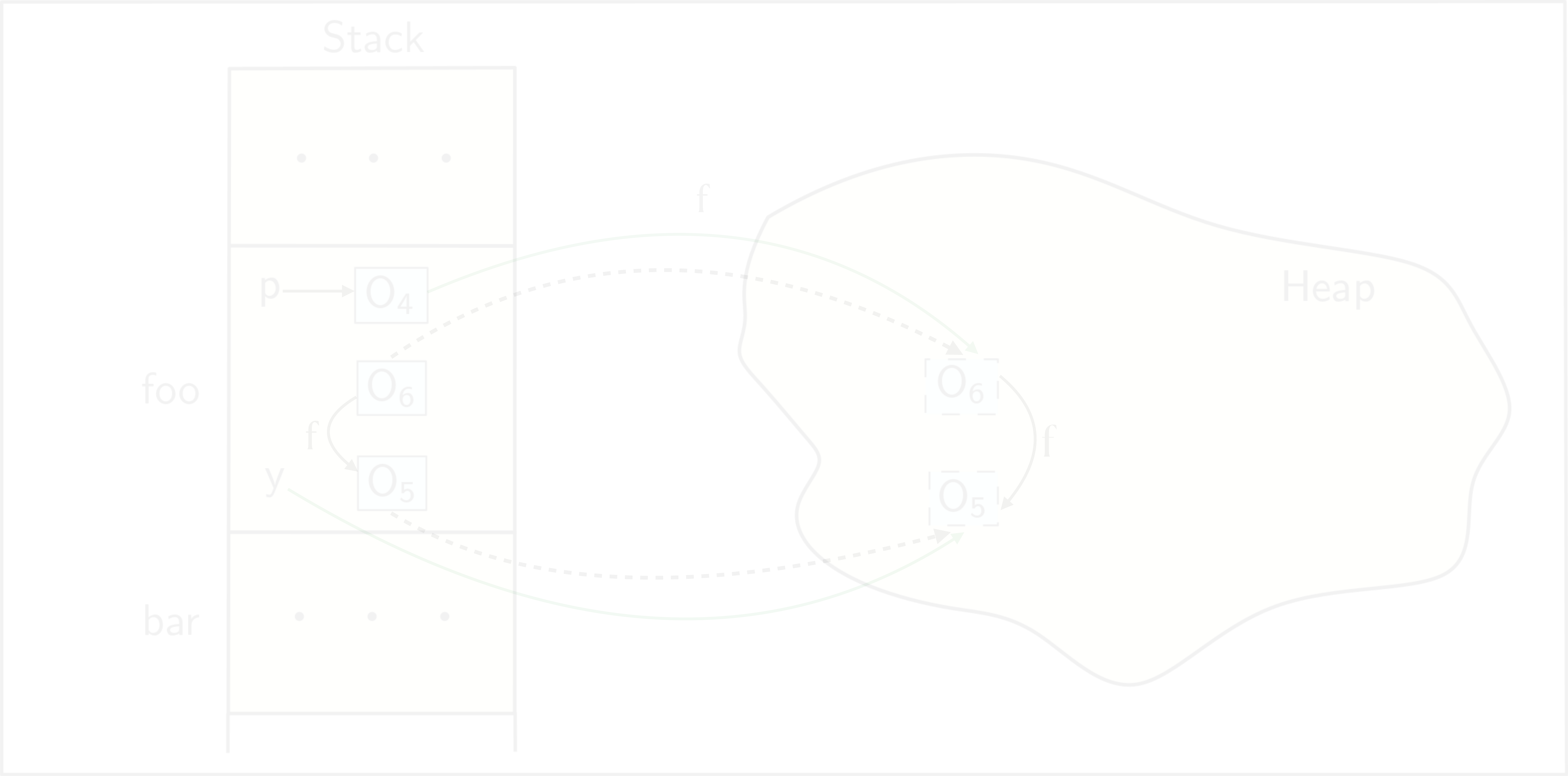
Heapification



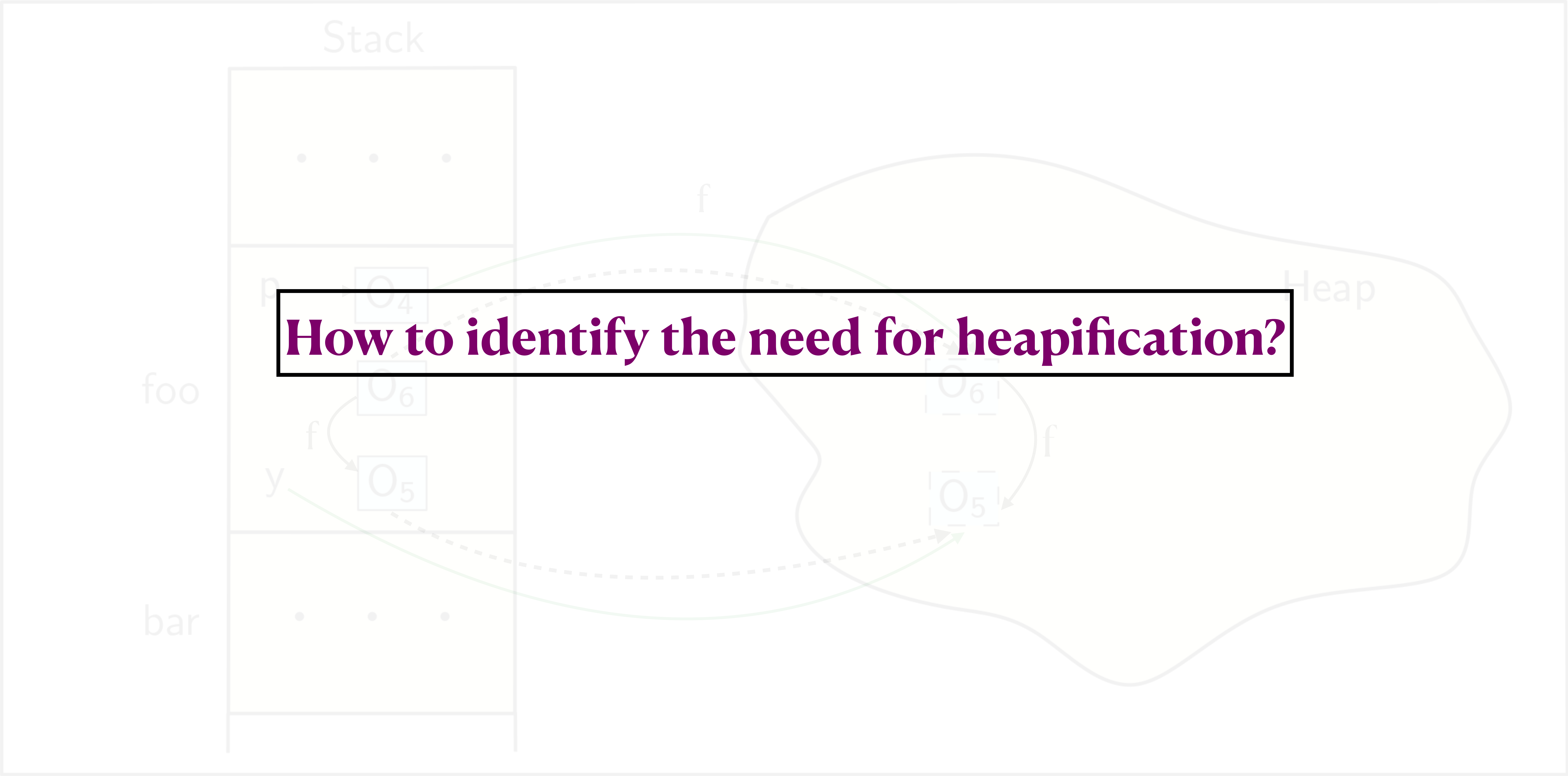
Heapification



Heapification



Heapification



How to identify the need for heapification?

Checking the Need for Heapification

Checking the Need for Heapification

- Dynamic heapification checks at each point where an object can escape:

Checking the Need for Heapification

- Dynamic heapification checks at each point where an object can escape:
    - Return of references. (Byte code: `return`.)

Checking the Need for Heapification

- Dynamic heapification checks at each point where an object can escape:
    - Return of references. (Byte code: `return.`)
    - References stores. (Byte code: `putfield, putstatic, aastore.`)

Checking the Need for Heapification

- Dynamic heapification checks at each point where an object can escape:
    - Return of references. (Byte code: `return.`)
    - References stores. (Byte code: `putfield, putstatic, aastore.`)
    - Throwing of exception. (Byte code: athrow`.`)

Checking the Need for Heapification

- Dynamic heapification checks at each point where an object can escape:
    - Return of references. (Byte code: `return.`)
    - References stores. (Byte code: `putfield, putstatic, aastore.`)
    - Throwing of exception. (Byte code: athrow.)
    - Calls to native. (Byte code: athrow.)

Checking the Need for Heapification

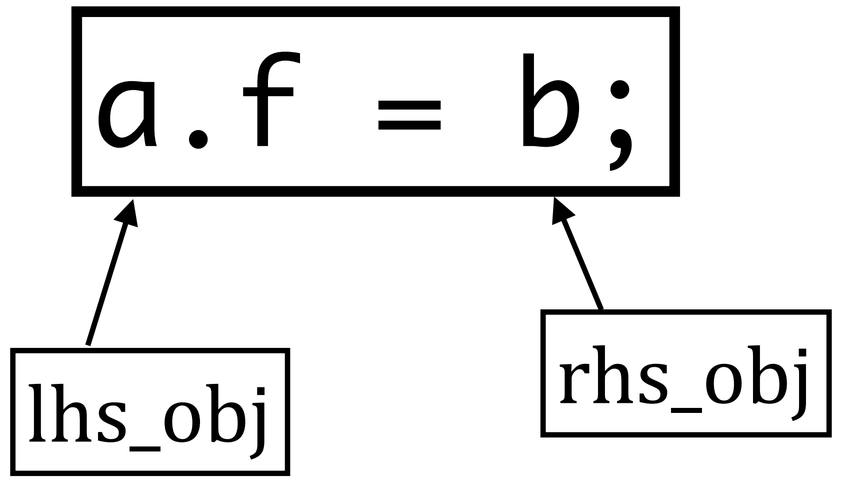
- Dynamic heapification checks at each point where an object can escape:
    - Return of references. (Byte code: `return.`)
    - References stores. (Byte code: `putfield, putstatic, aastore.`)
    - Throwing of exception. (Byte code: athrow.)
    - Calls to native. (Byte code: athrow.)
    - JNI APIs 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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```

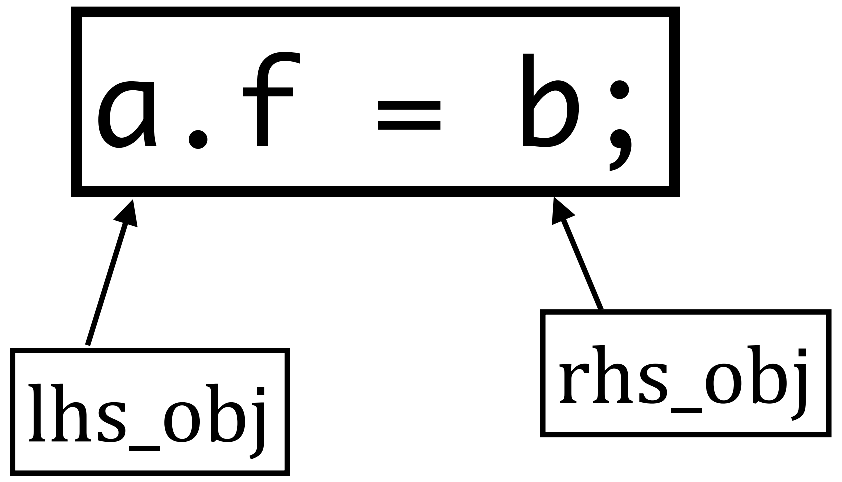
Checking the Need for Heapification

```
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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



Checking the Need for Heapification

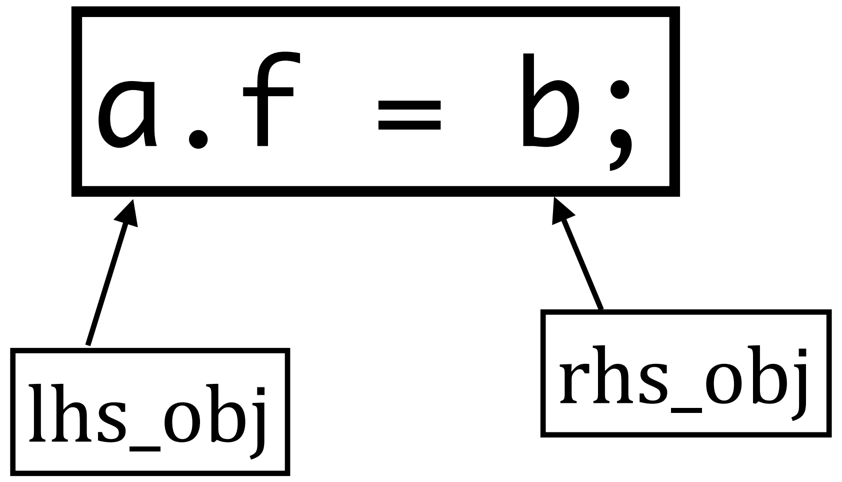
```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



`a.f = b;`

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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



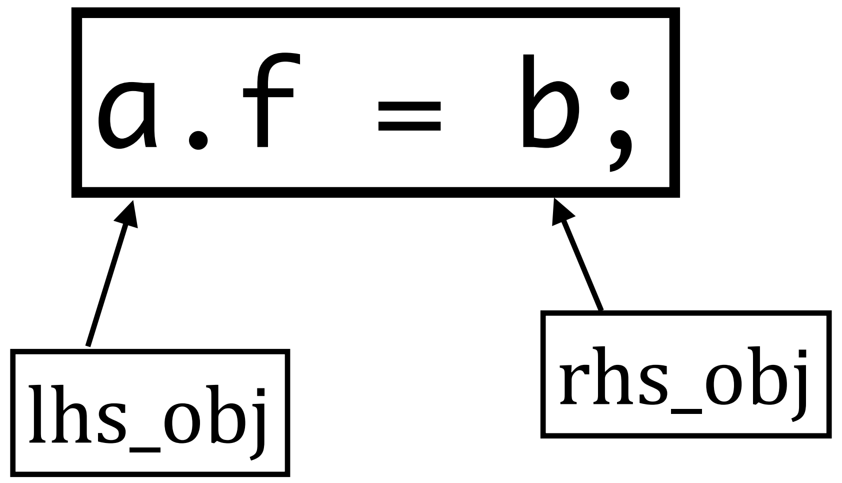
`a.f = b;`

lhs_obj

rhs_obj

Checking the Need for Heapification

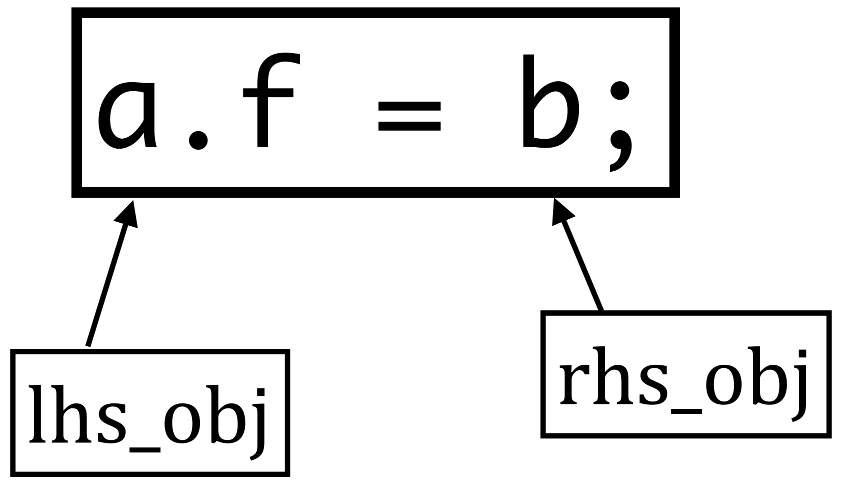
```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



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

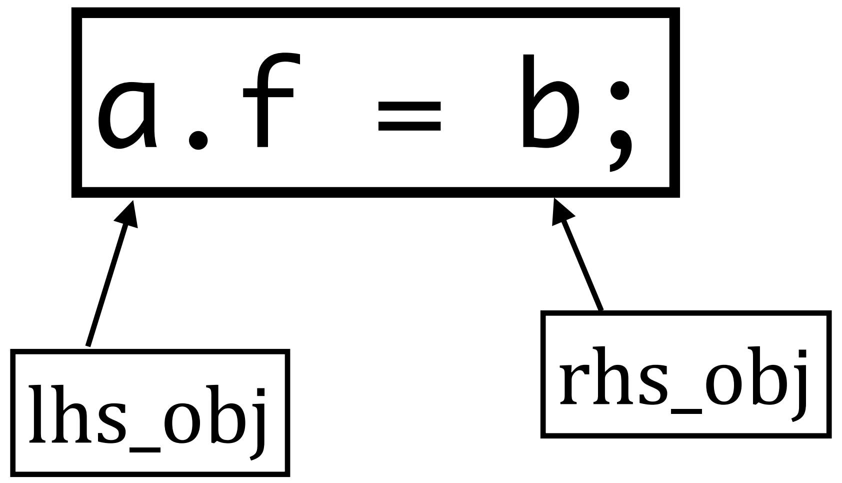
Checking the Need for Heapification

```
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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



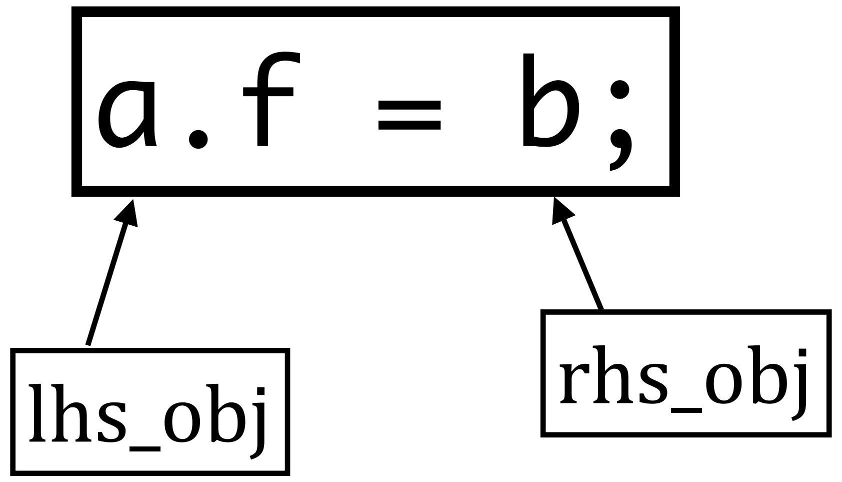
a.f = b;

lhs_obj

rhs_obj

Checking the Need for Heapification

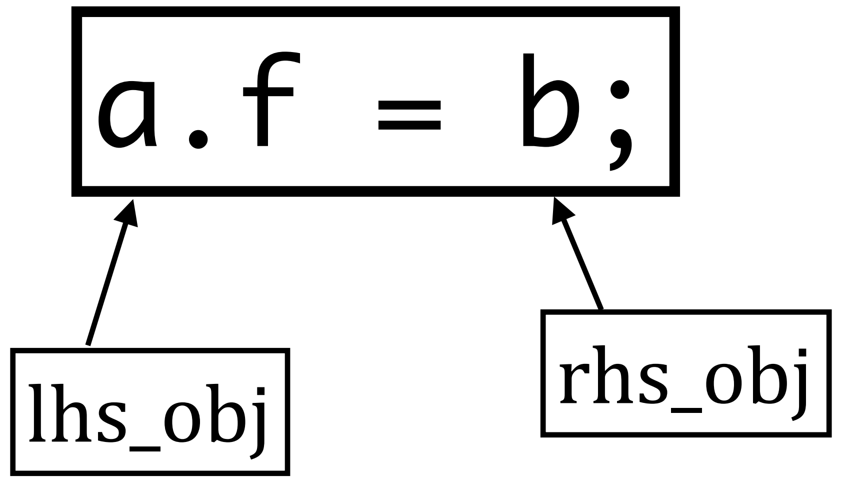
```
Procedure HeapificationCheckAtStore(lhs, rhs)
    if rhs object is outside stack bounds then
        No heapification required.
    else
        /* The rhs object is present on the stack */
        if lhs object is outside stack bounds then
            Heapify starting from the rhs object.
        else
            /* Both lhs and rhs objects are on the stack */
            if rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



```
rhs_obj >= lhs_obj
```

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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```

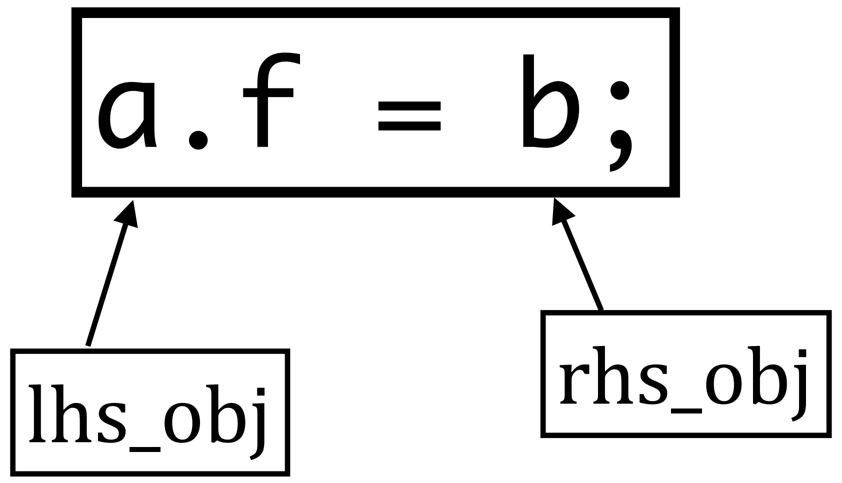


```
a.f = b;
```

```
rhs_obj >= lhs_obj
```

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 rhs object has been allocated before the lhs object then
                No heapification required.
            else
                /* The lhs object has been allocated in either the same frame or a deeper frame as
                   compared to the rhs object */
                Perform stack-walk and heapify if needed.
```



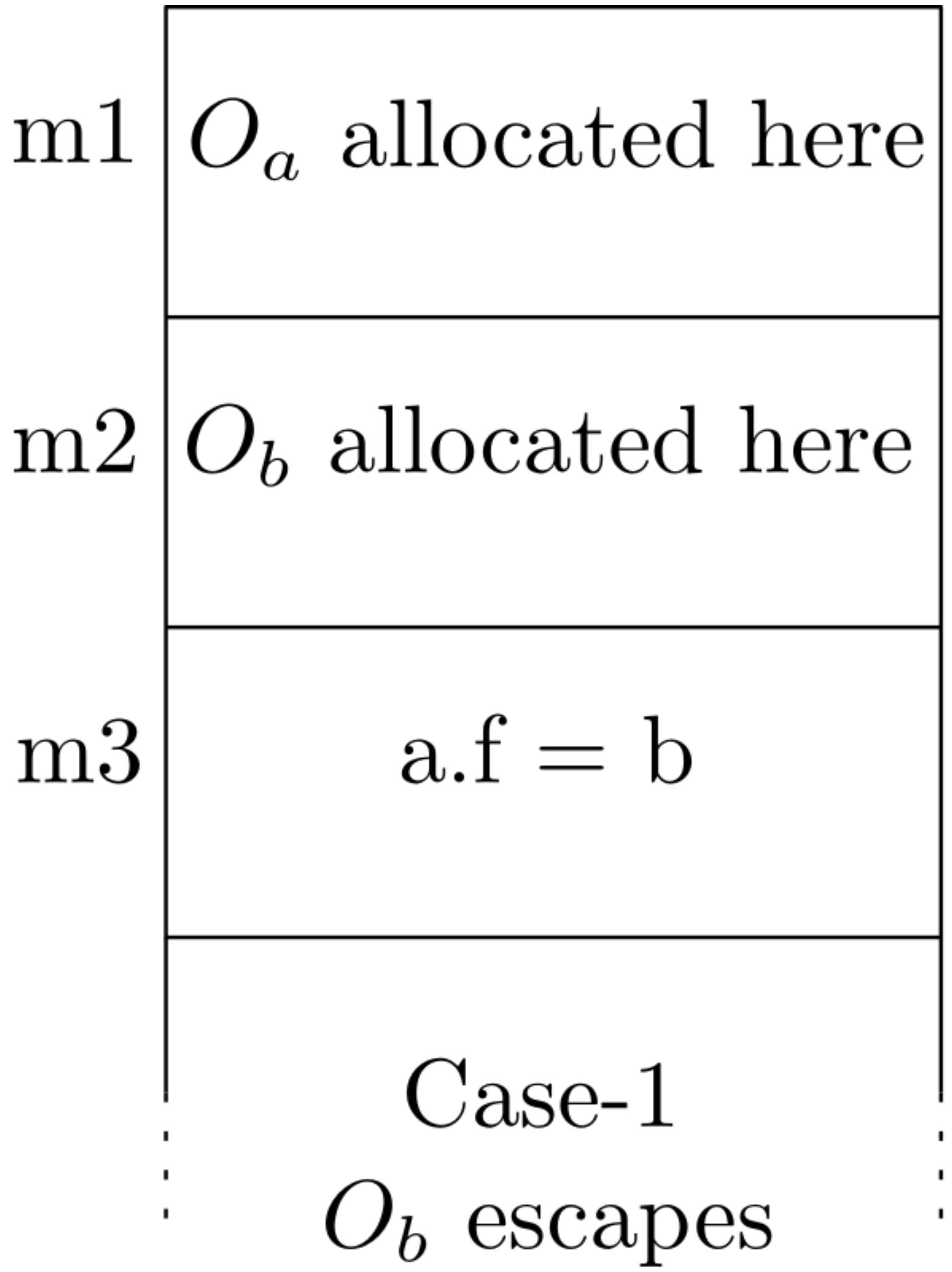
`rhs_obj >= lhs_obj`

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 */
```



m1 O_a allocated here

m2 O_b allocated here

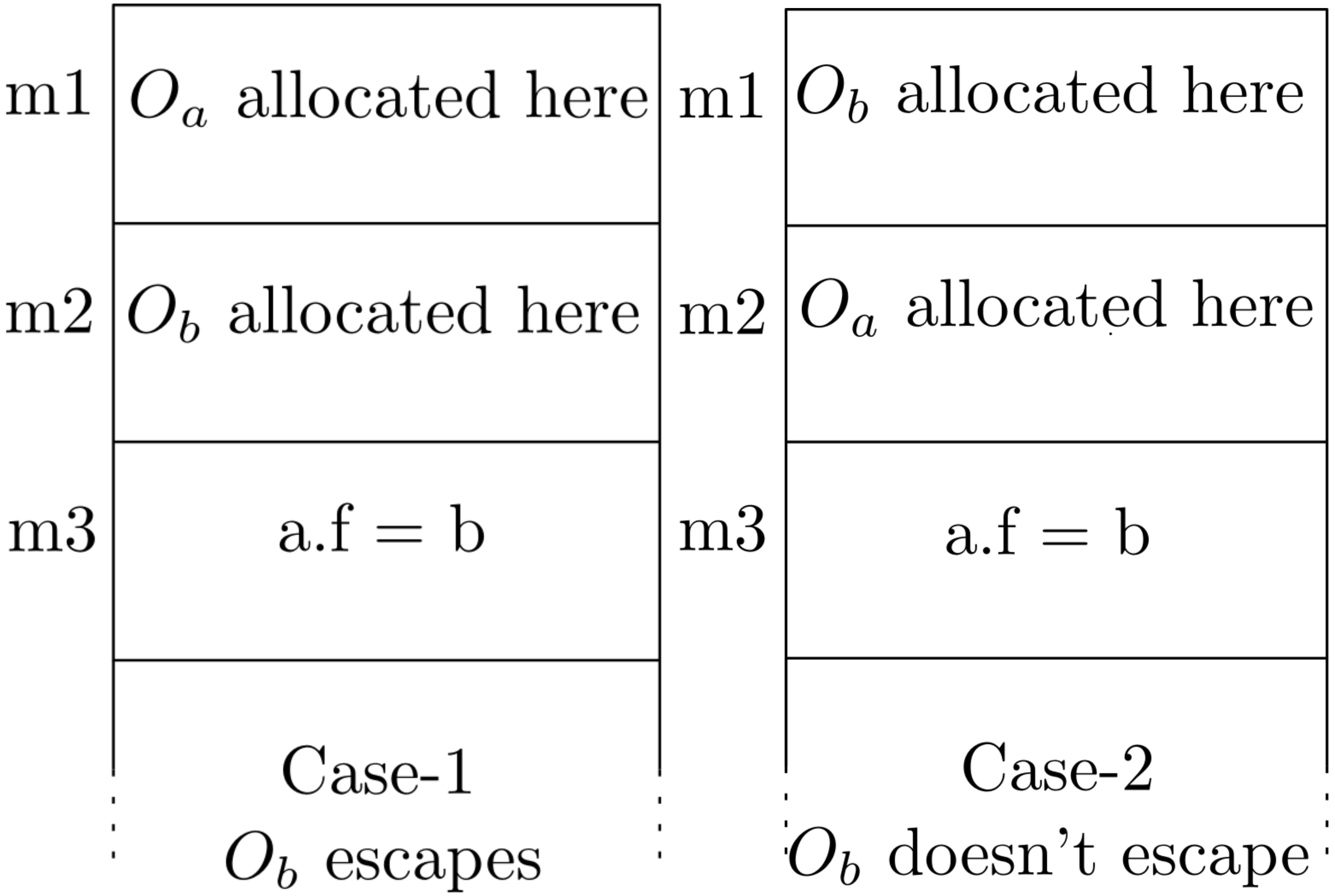
m3 a.f = b

Case-1

O_b escapes

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 */
```



m1 O_a allocated here

m2 O_b allocated here

m3 a.f = b

Case-1

O_b escapes

m1 O_b allocated here

m2 O_a allocated here

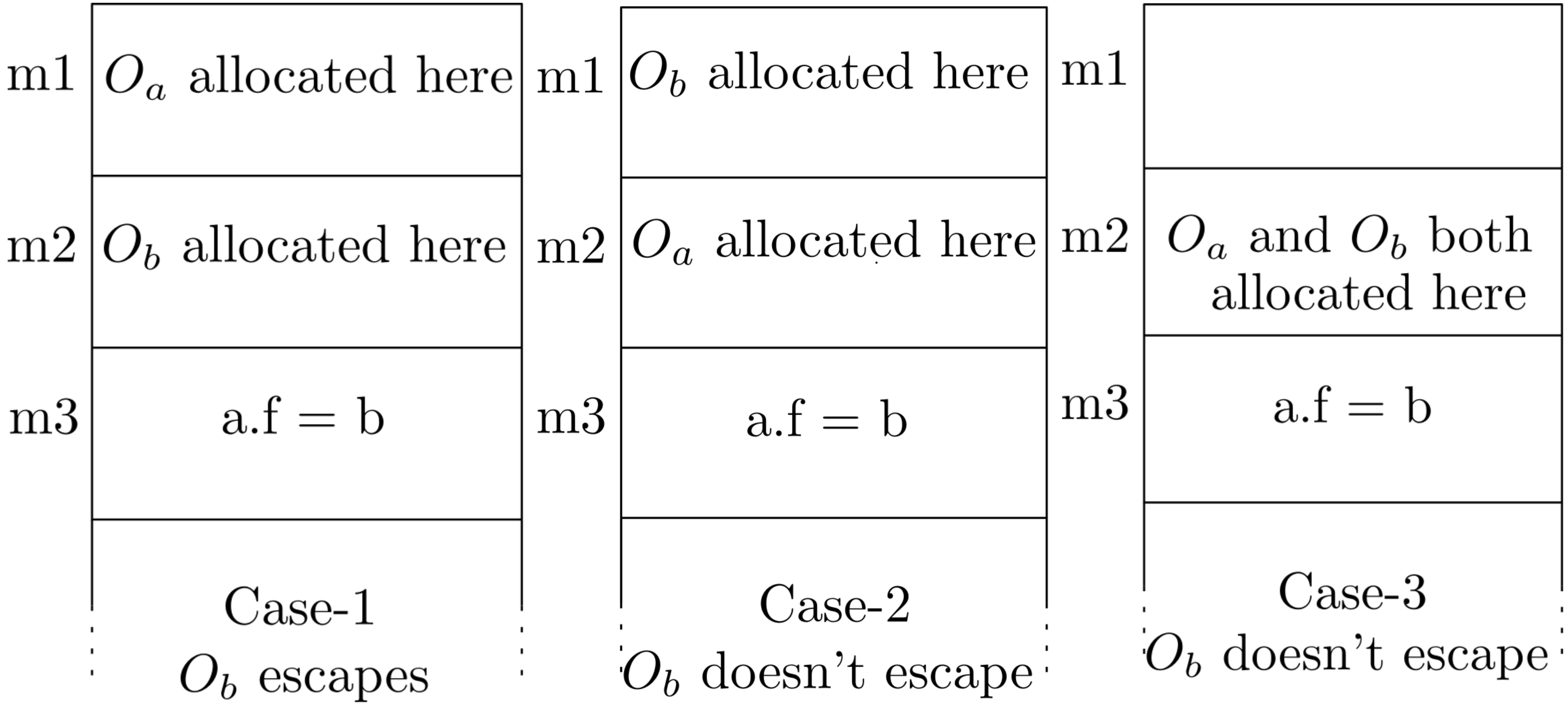
m3 a.f = b

Case-2

O_b doesn't escape

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 */
```



m1 O_a allocated here

m2 O_b allocated here

m3 a.f = b

Case-1

O_b escapes

m1 O_b allocated here

m2 O_a allocated here

m3 a.f = b

Case-2

O_b doesn't escape

m1

m2 O_a and O_b both allocated here

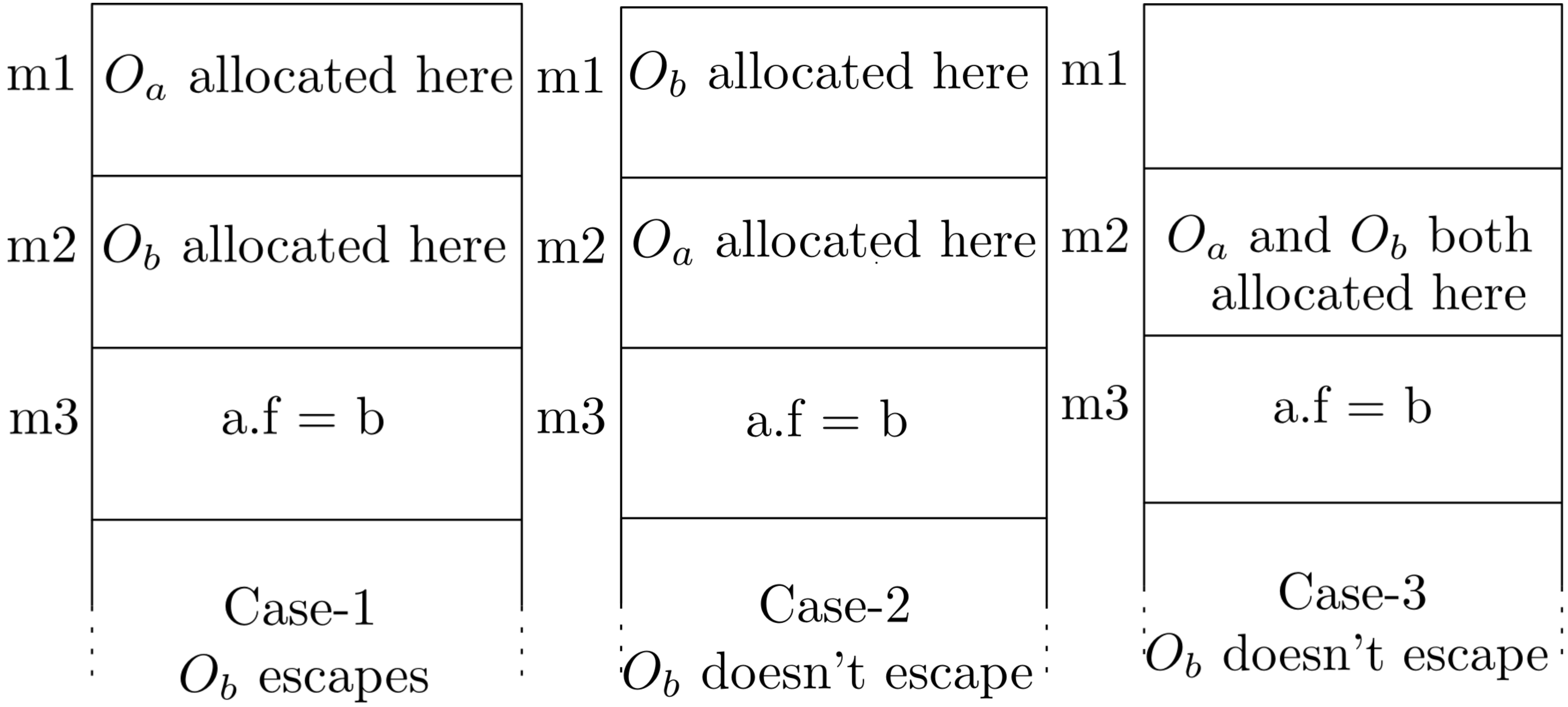
m3 a.f = b

Case-3

O_b doesn't escape

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

Ordering Objects on Stack

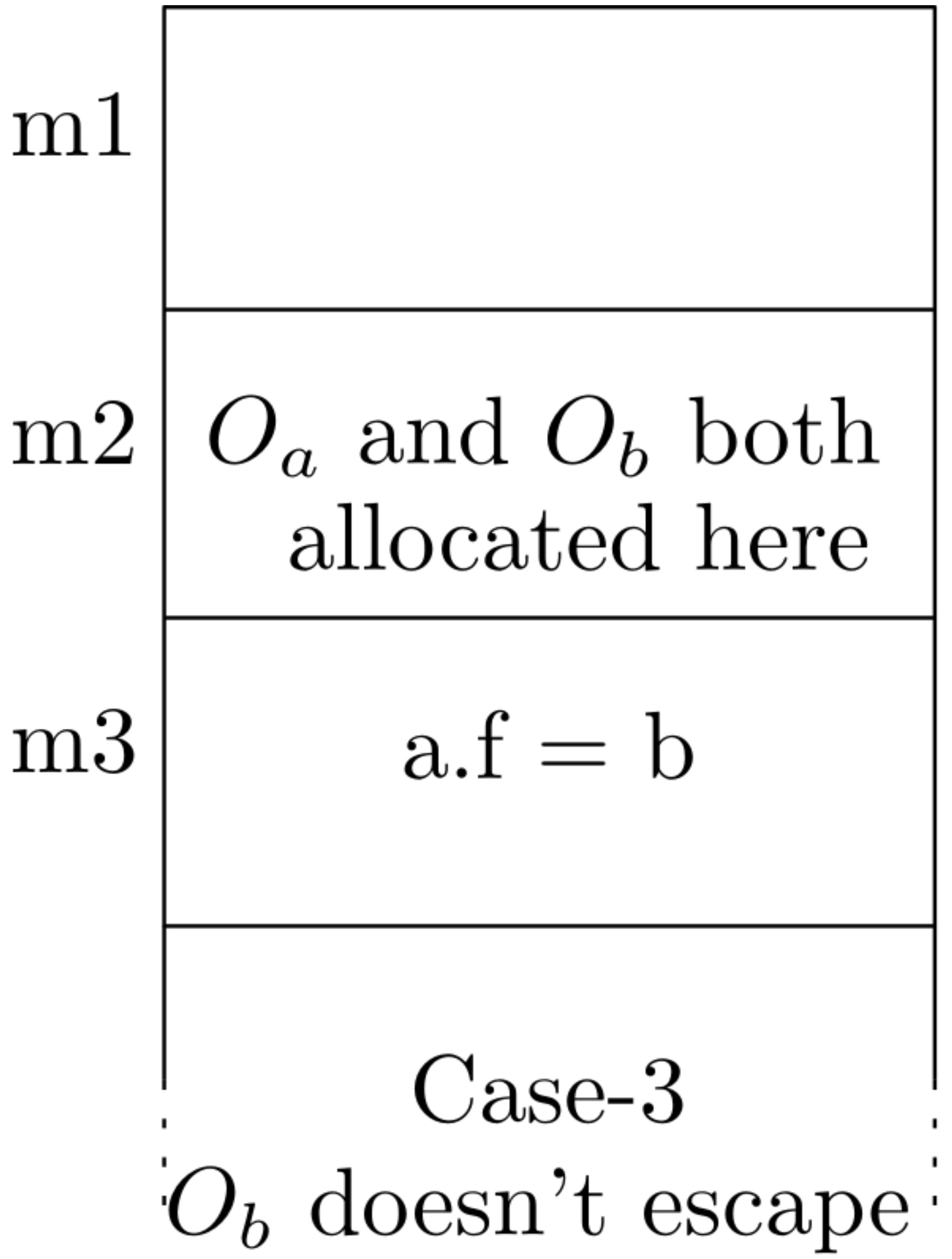
Ordering Objects on Stack

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.



m1

m2 O_a and O_b both allocated here

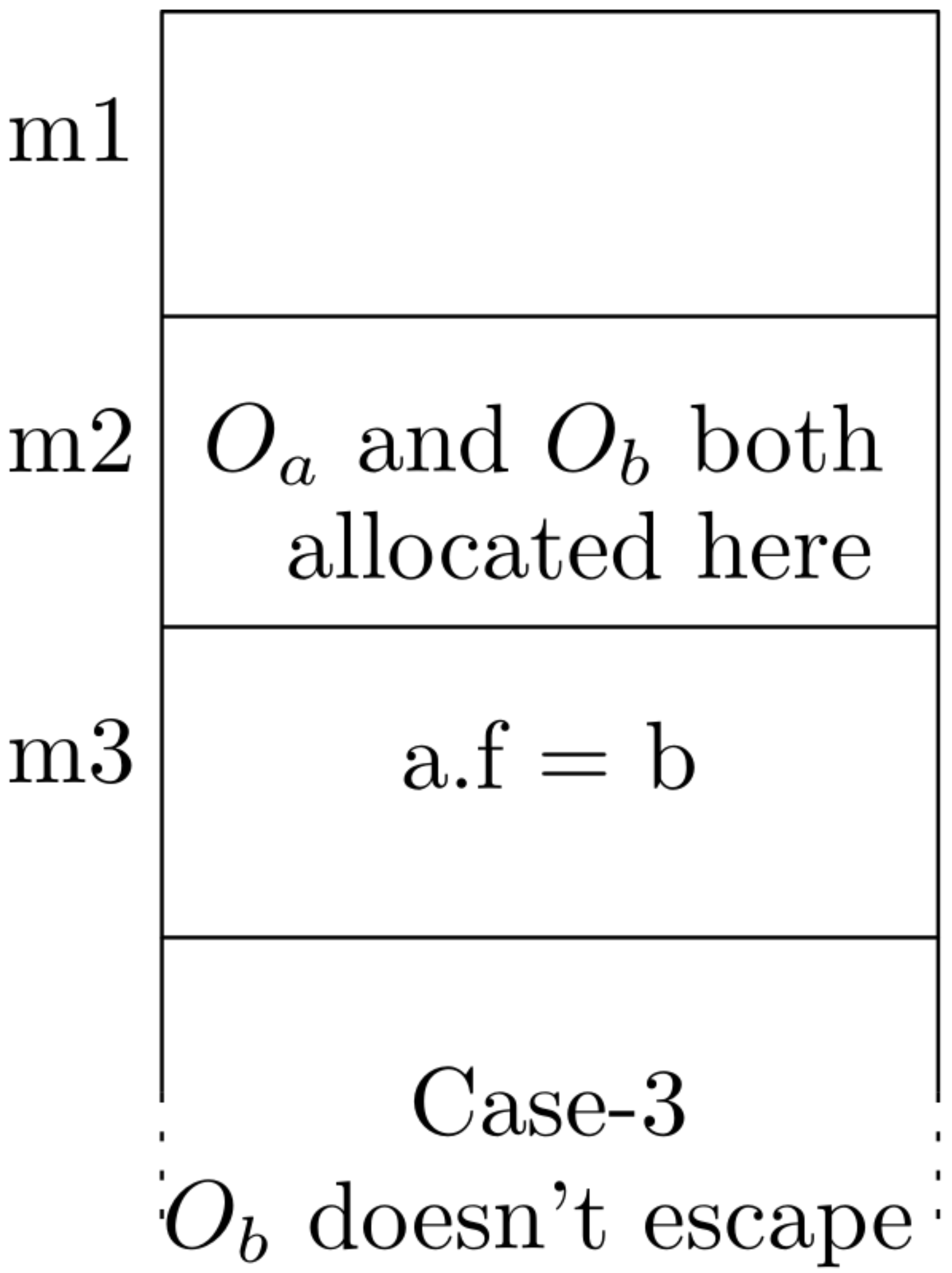
m3 a.f = b

Case-3

O_b doesn't escape

Ordering Objects on Stack

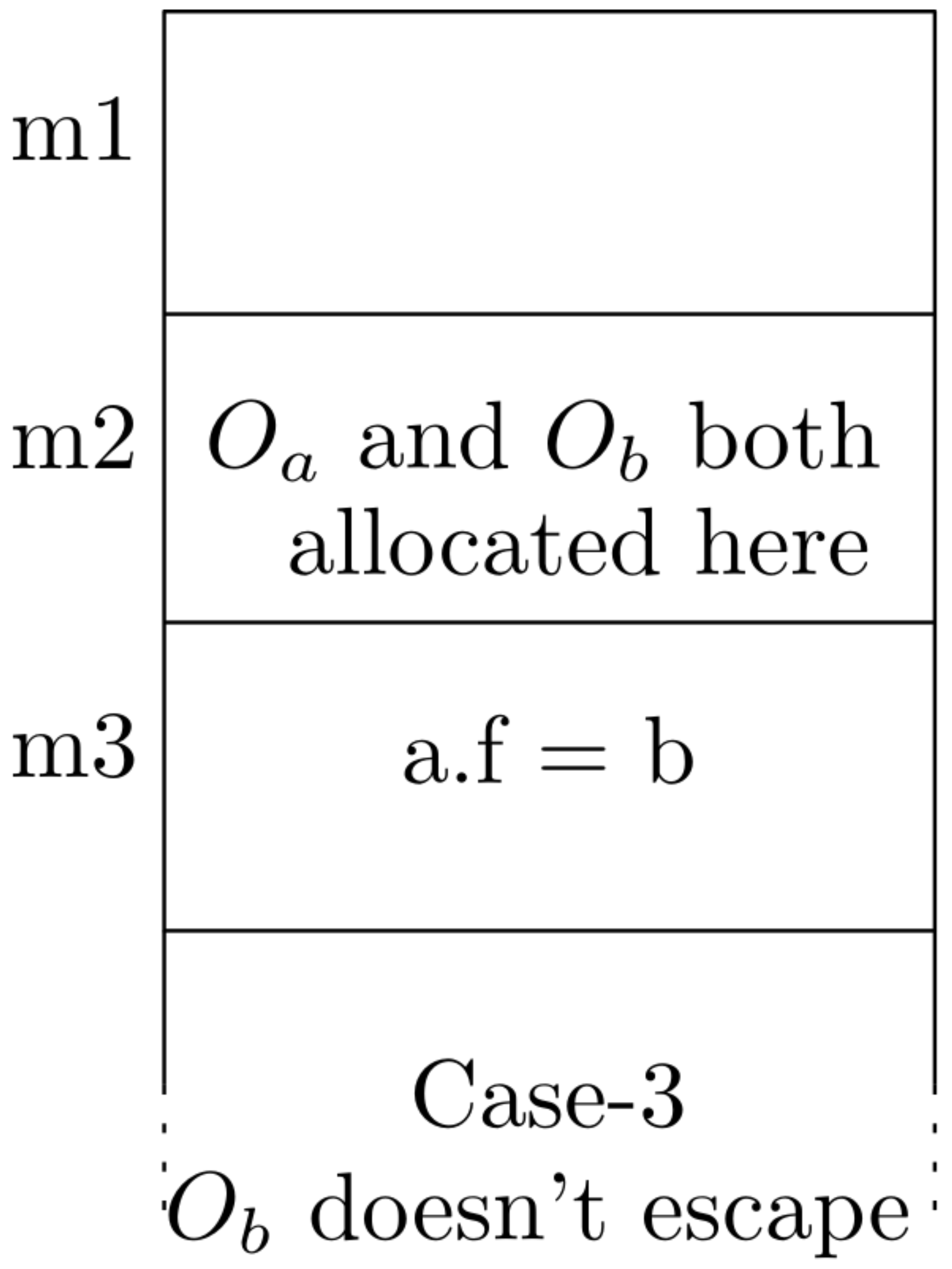
- A simple address-comparison check works majority of times.



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

Ordering Objects on Stack

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



m1

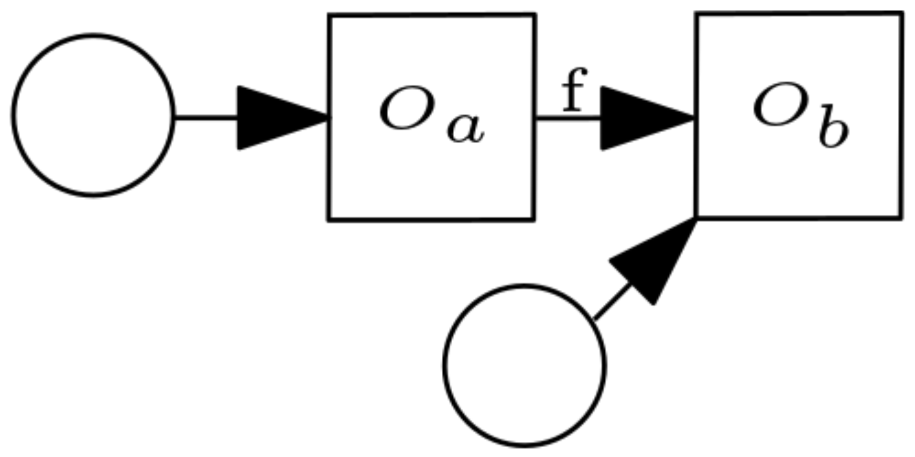
m2 O_a and O_b both allocated here

m3 a.f = b

Case-3

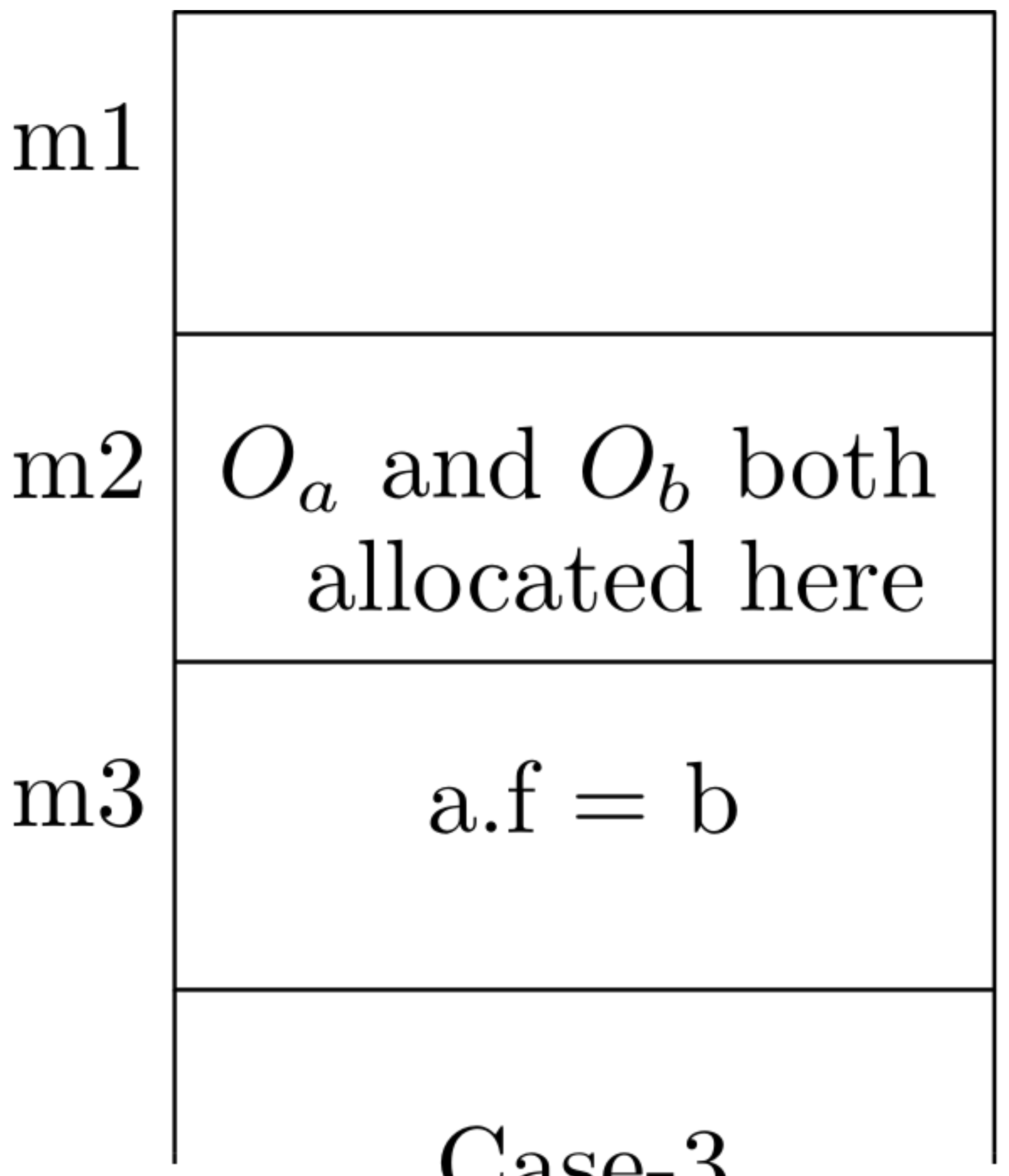
O_b doesn't escape

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



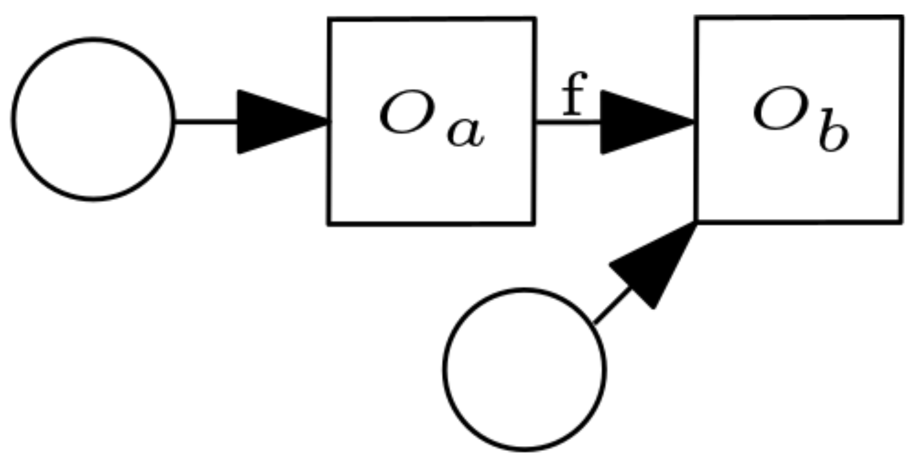
Ordering Objects on Stack

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



Case-3
O_b doesn't escape

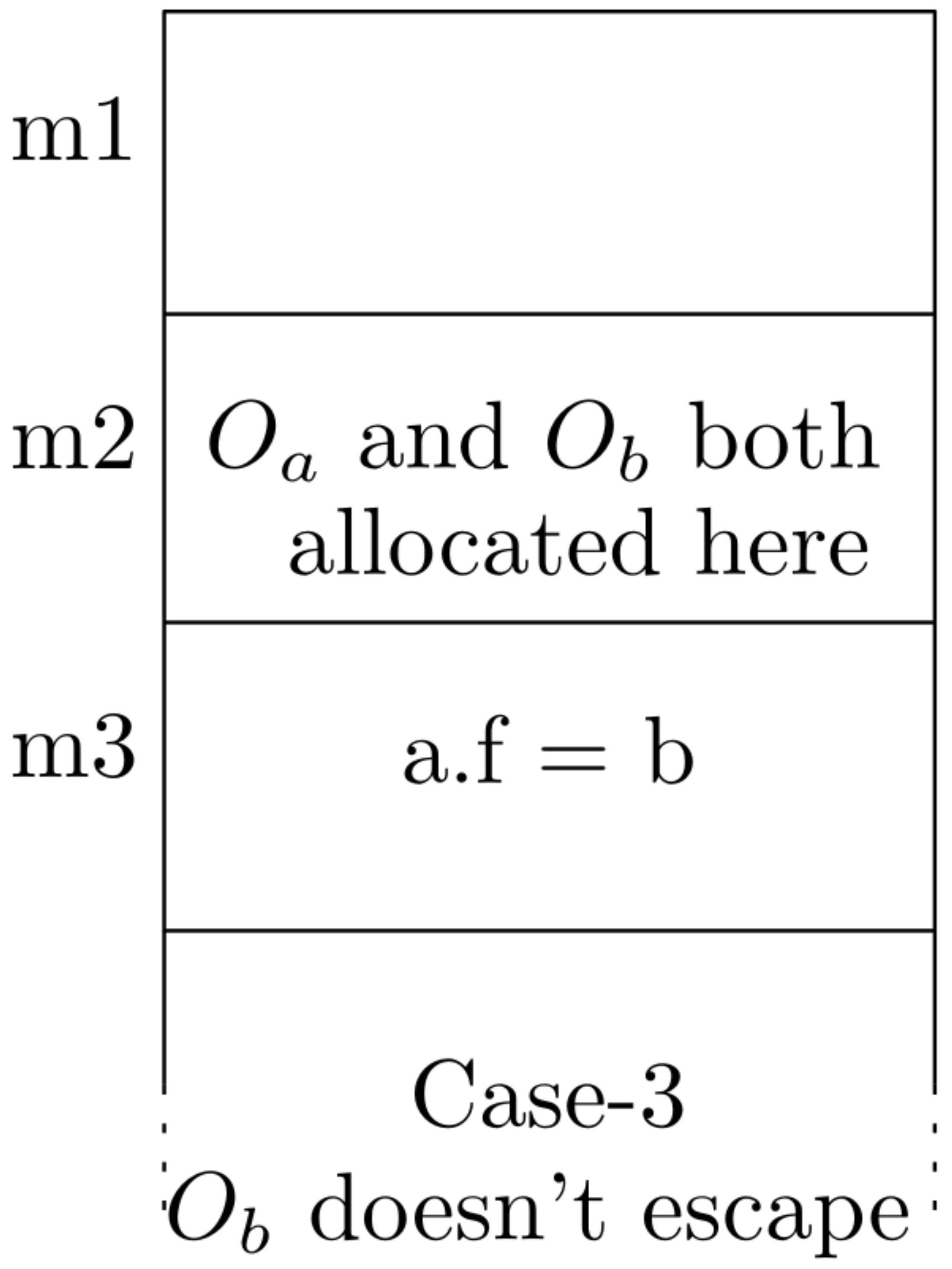
- Statically create a partial order of stack-allocatable objects.



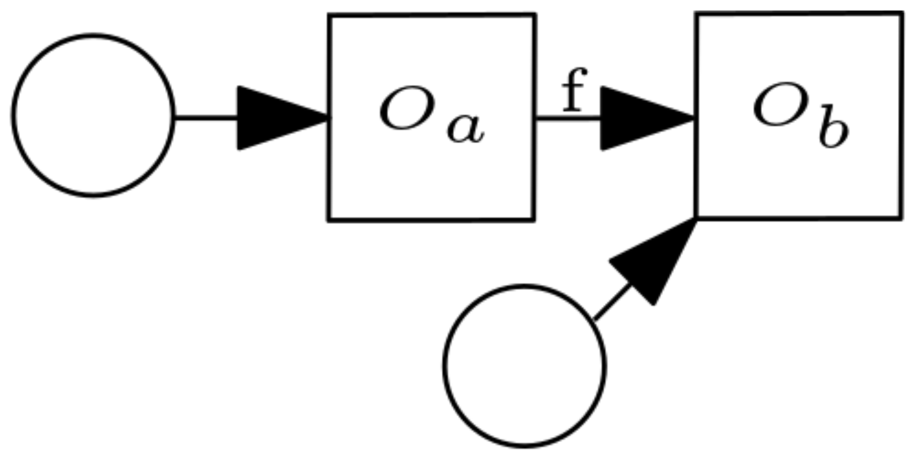
[O_b, O_a]

Ordering Objects on Stack

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



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

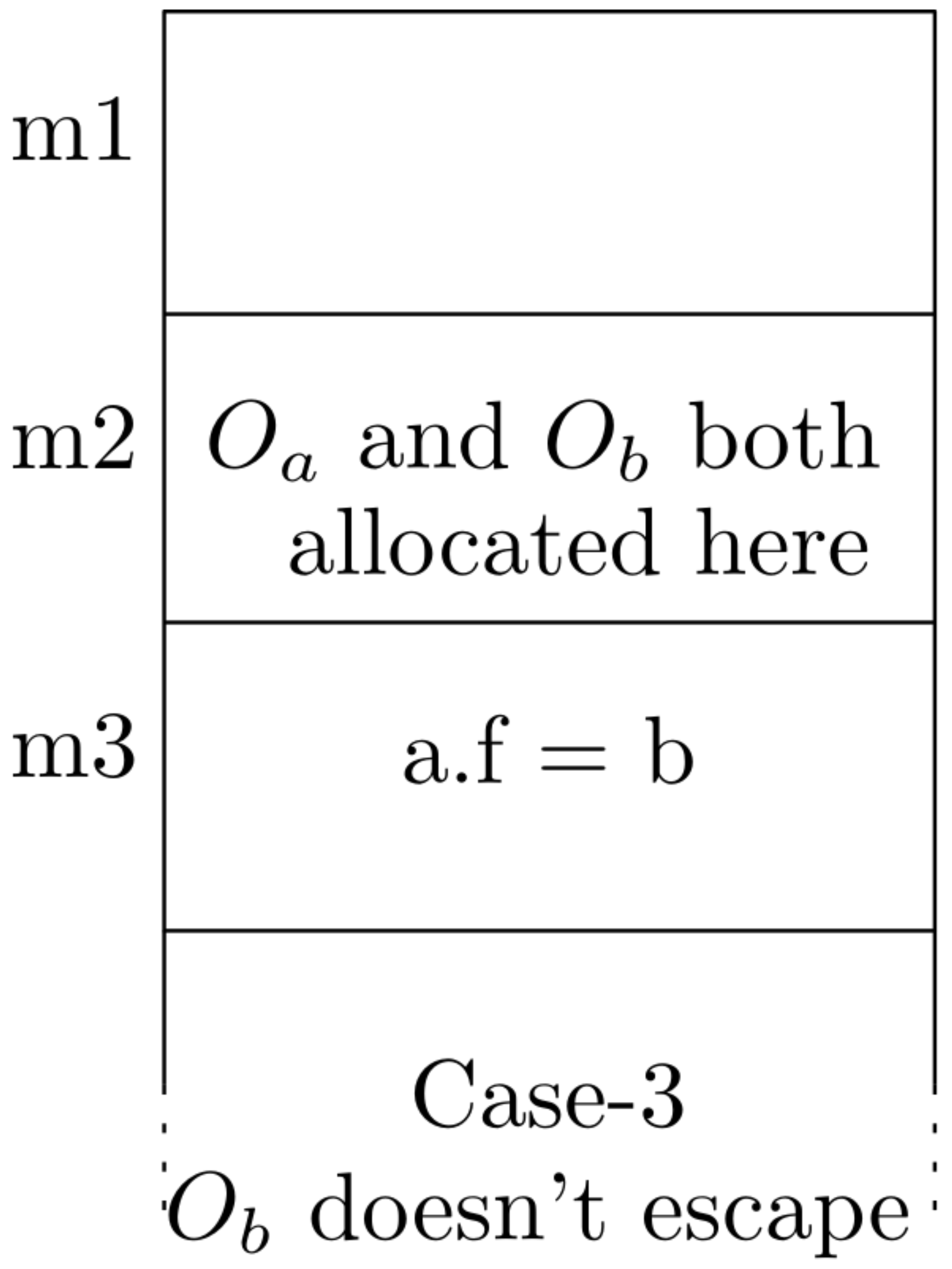


$[O_b, O_a]$

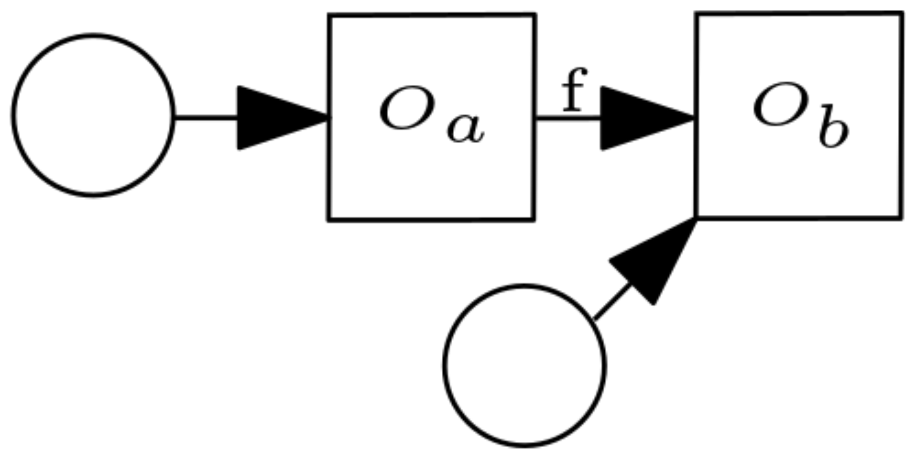
- Use the stack-order in VM to re-order the list of stack allocated objects.

Ordering Objects on Stack

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



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

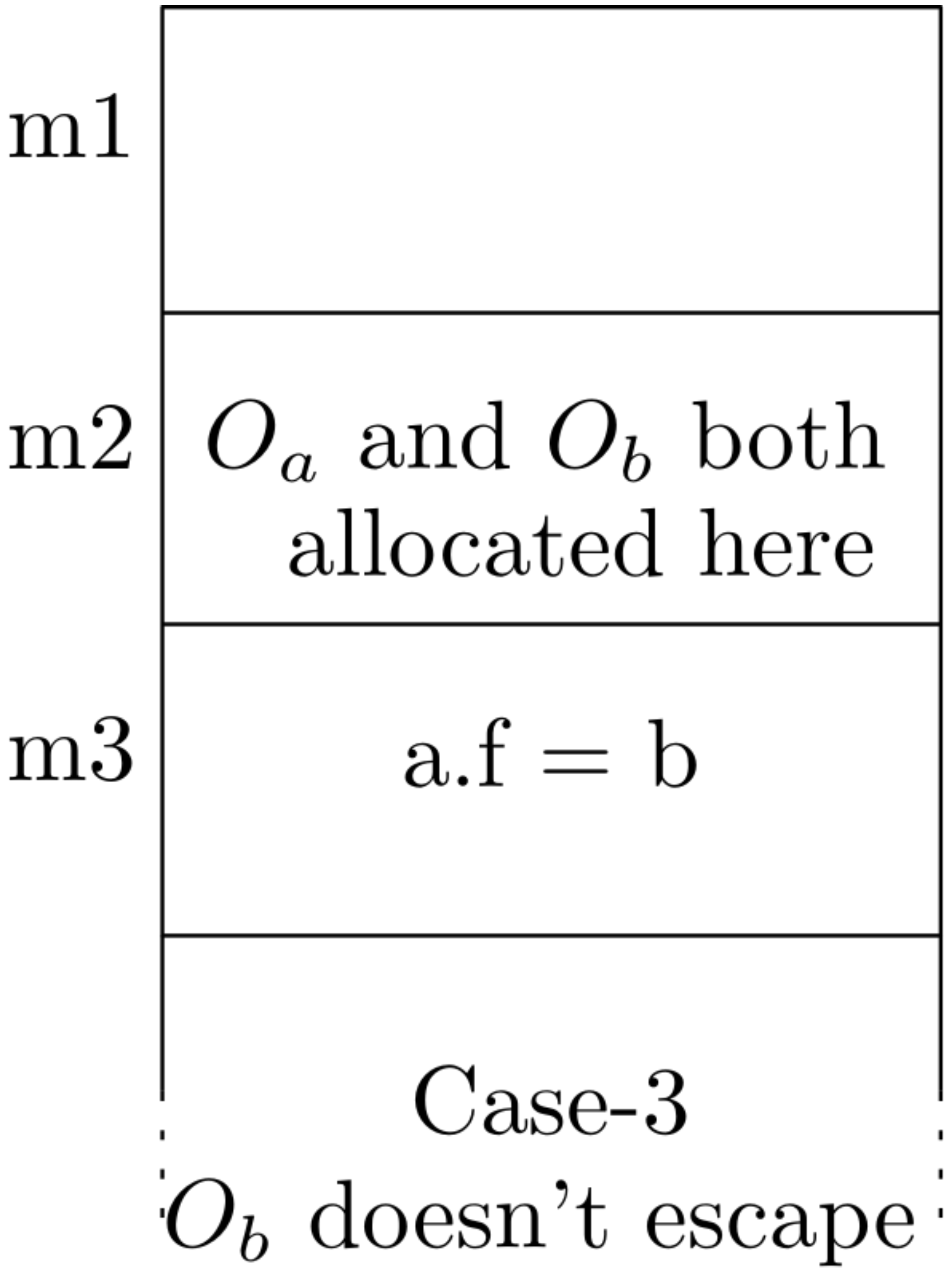


[O_b, O_a]

- 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.



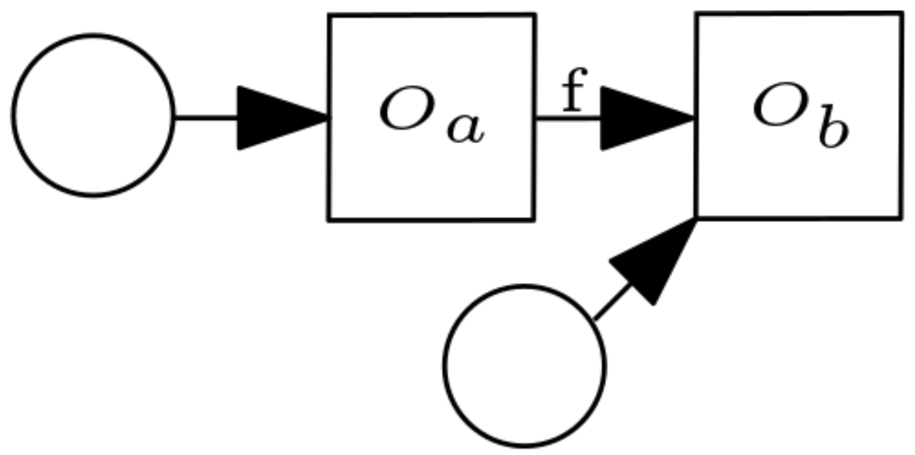
m1

m2 O_a and O_b both allocated here

m3 a.f = b

Case-3
O_b doesn't escape

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

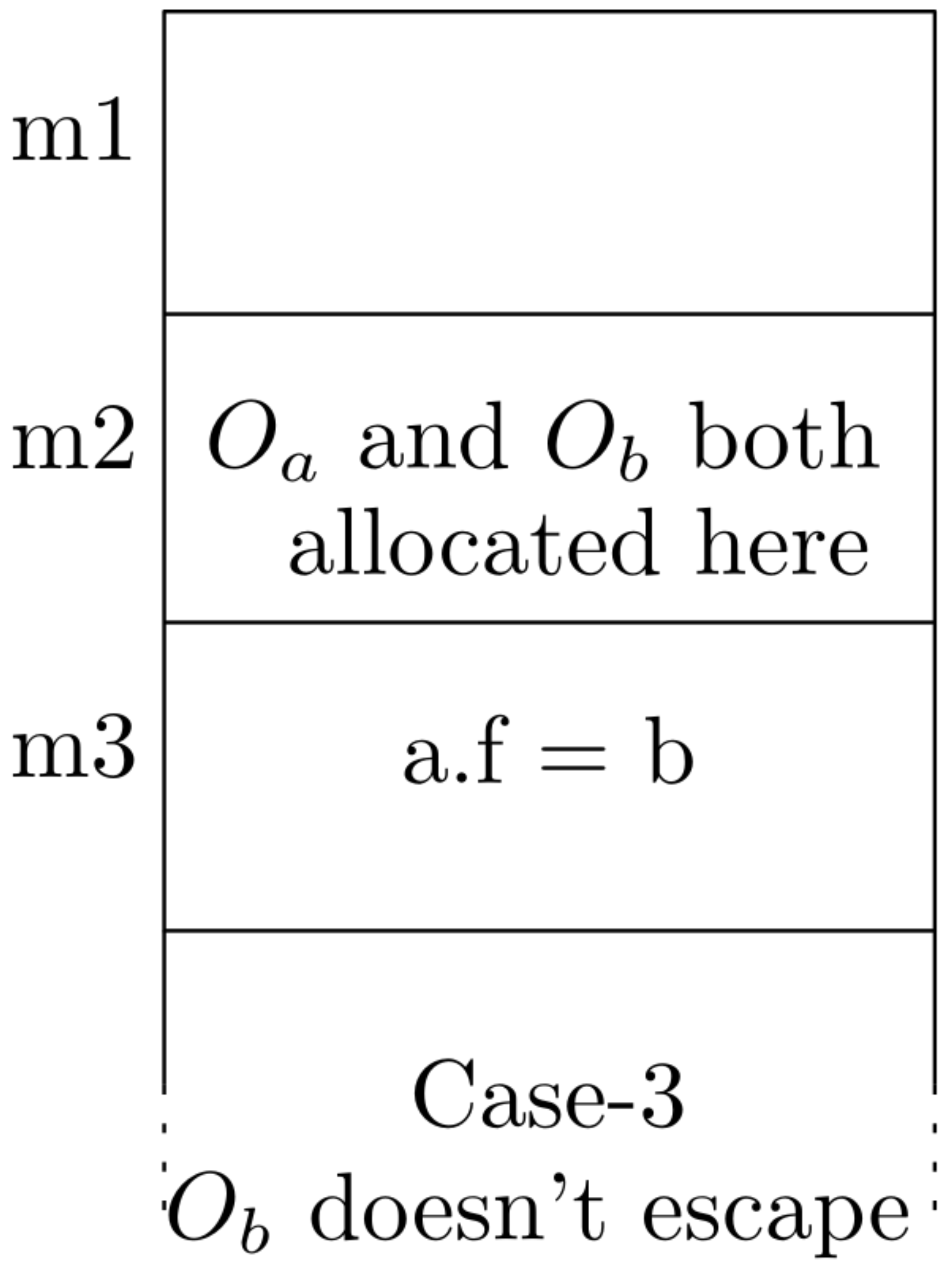


[O_b, O_a]

- 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.



m1

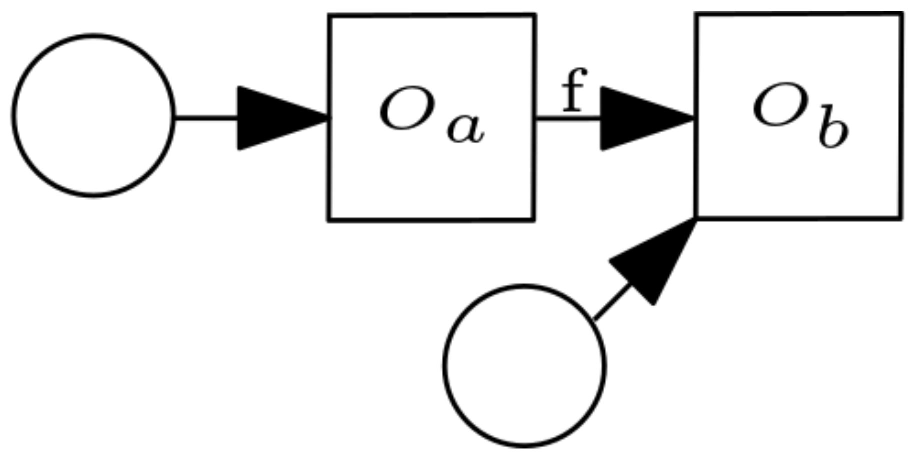
m2 O_a and O_b both allocated here

m3 a.f = b

Case-3

O_b doesn't escape

- 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

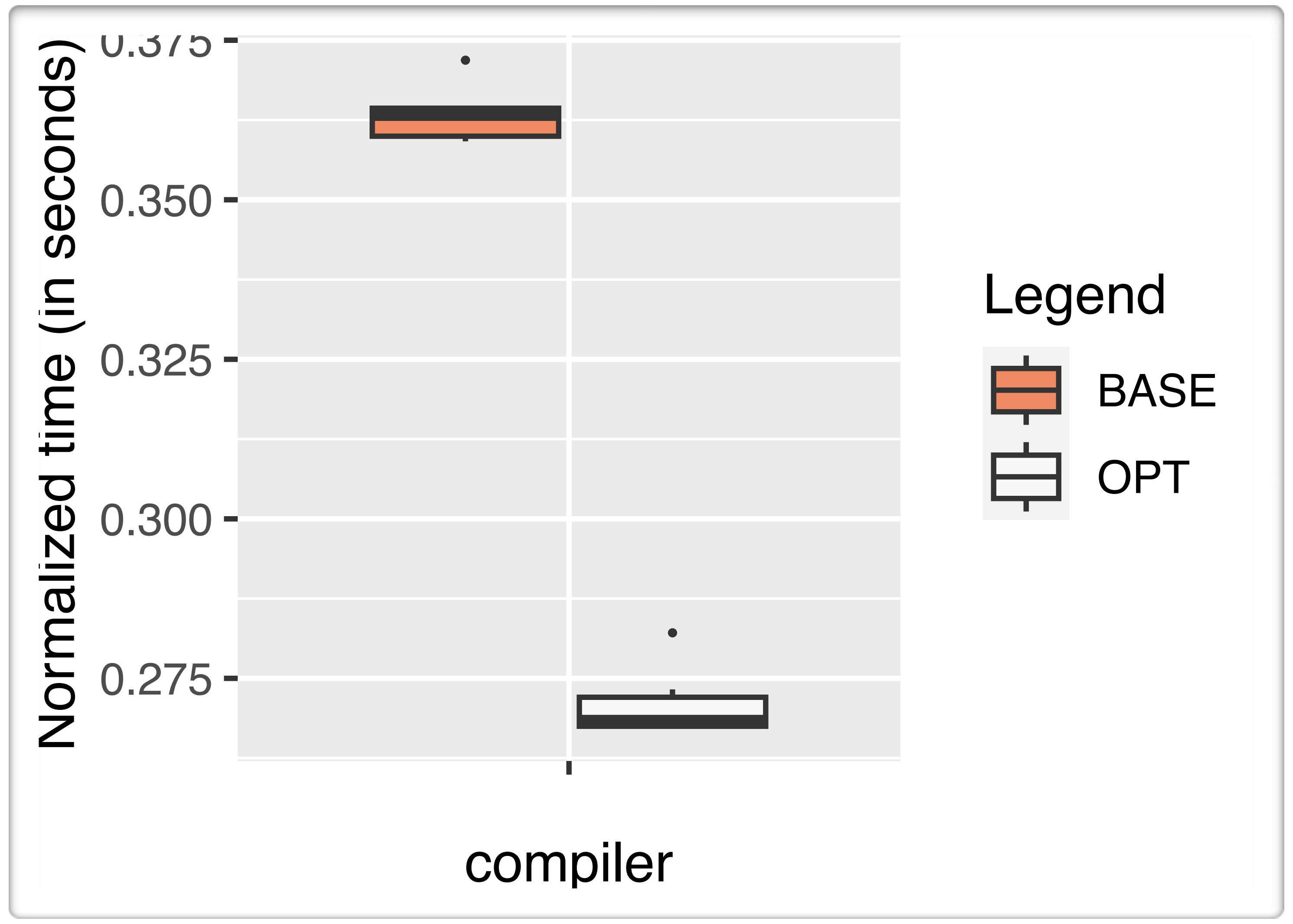
Evaluation (Stack Allocation)

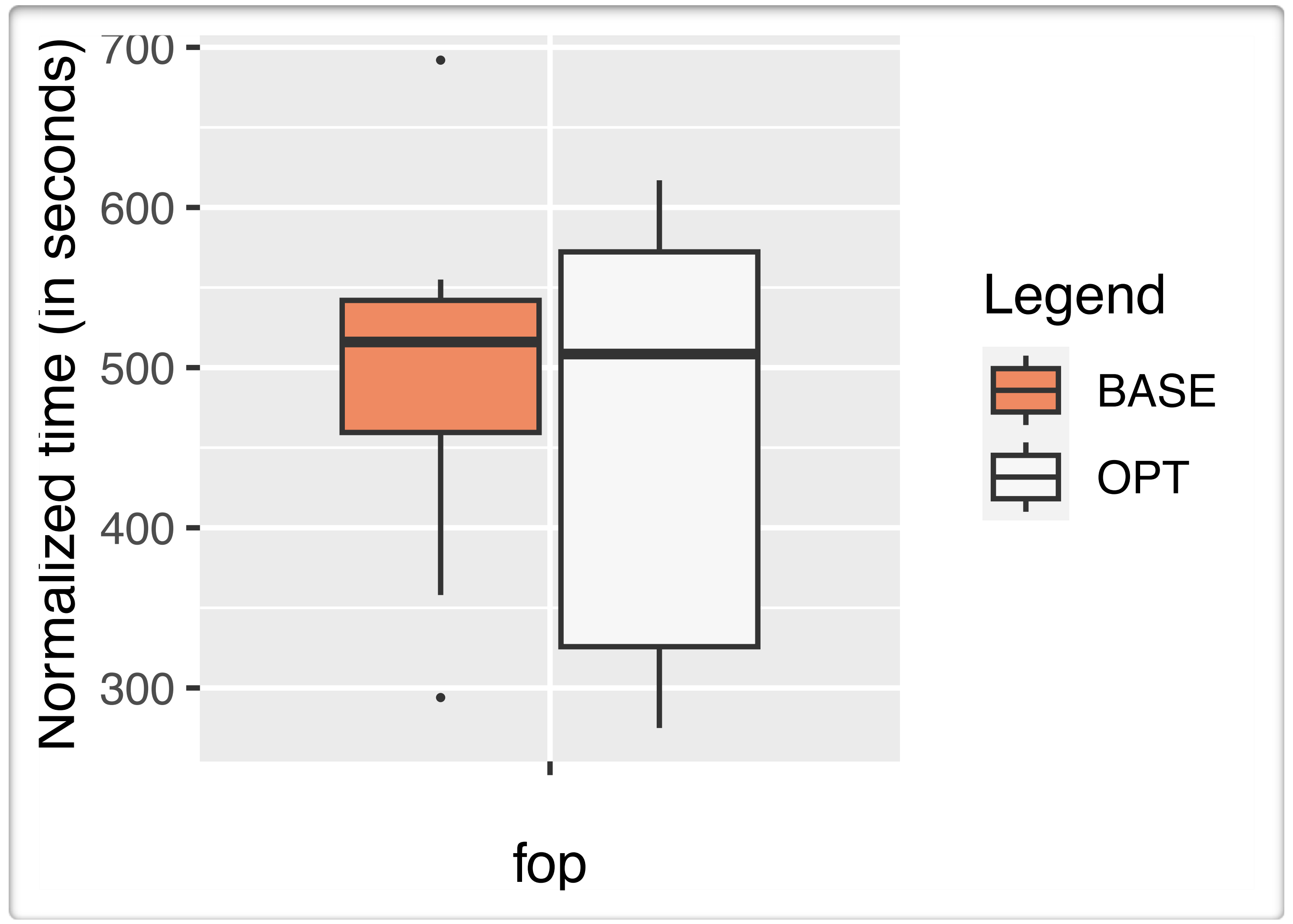
	Non Optimistic Scheme (BASE)			Optimistic Scheme (OPT)		
Benchmark	Static Count	Dynamic Count	Stack Bytes	Static Count	Dynamic Count	Stack Bytes
graphchi	0 (0.0 %)	0M (0.00%)	0MB	32 (4.15%)	506.3M (6.9%)	9184.6MB
fop	10 (0.15%)	0.04M (0.002%)	1MB	50 (0.77%)	9.8M (0.42%)	161.2MB
h2	61 (2.33%)	29M (0.92%)	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

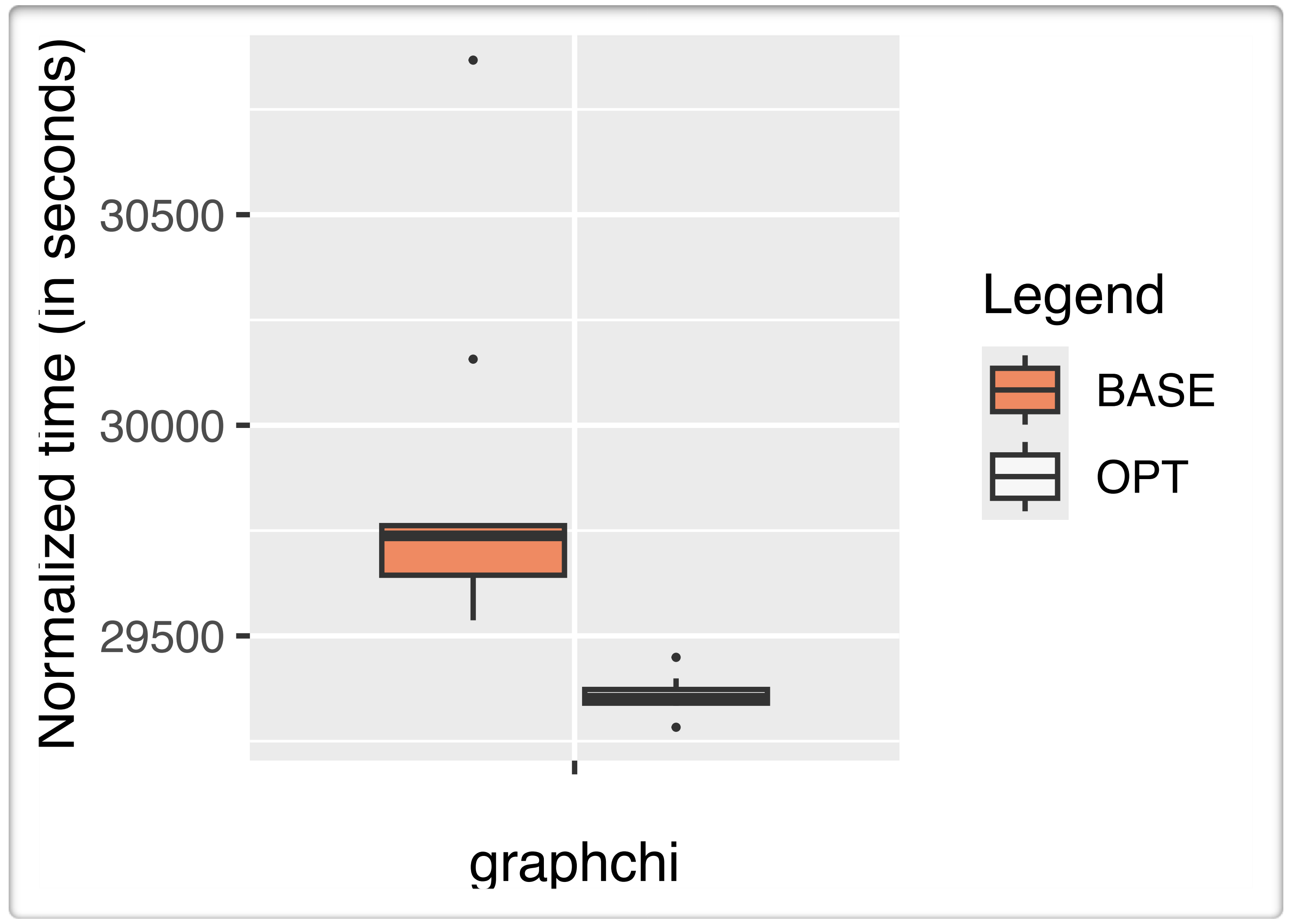
Stack Allocation: 71%↑ Stack Bytes: 54%↑

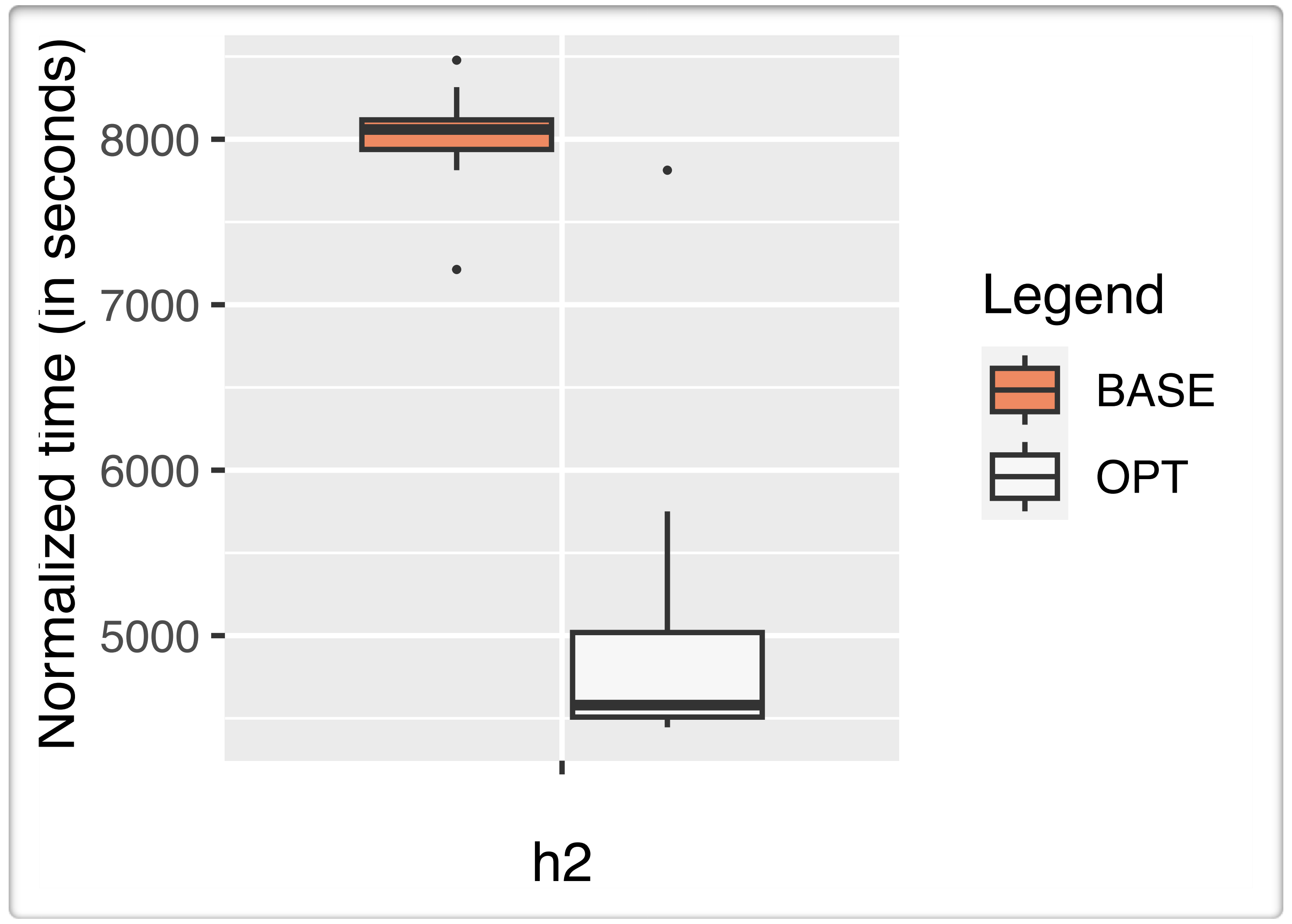
(Less Heap Allocation)

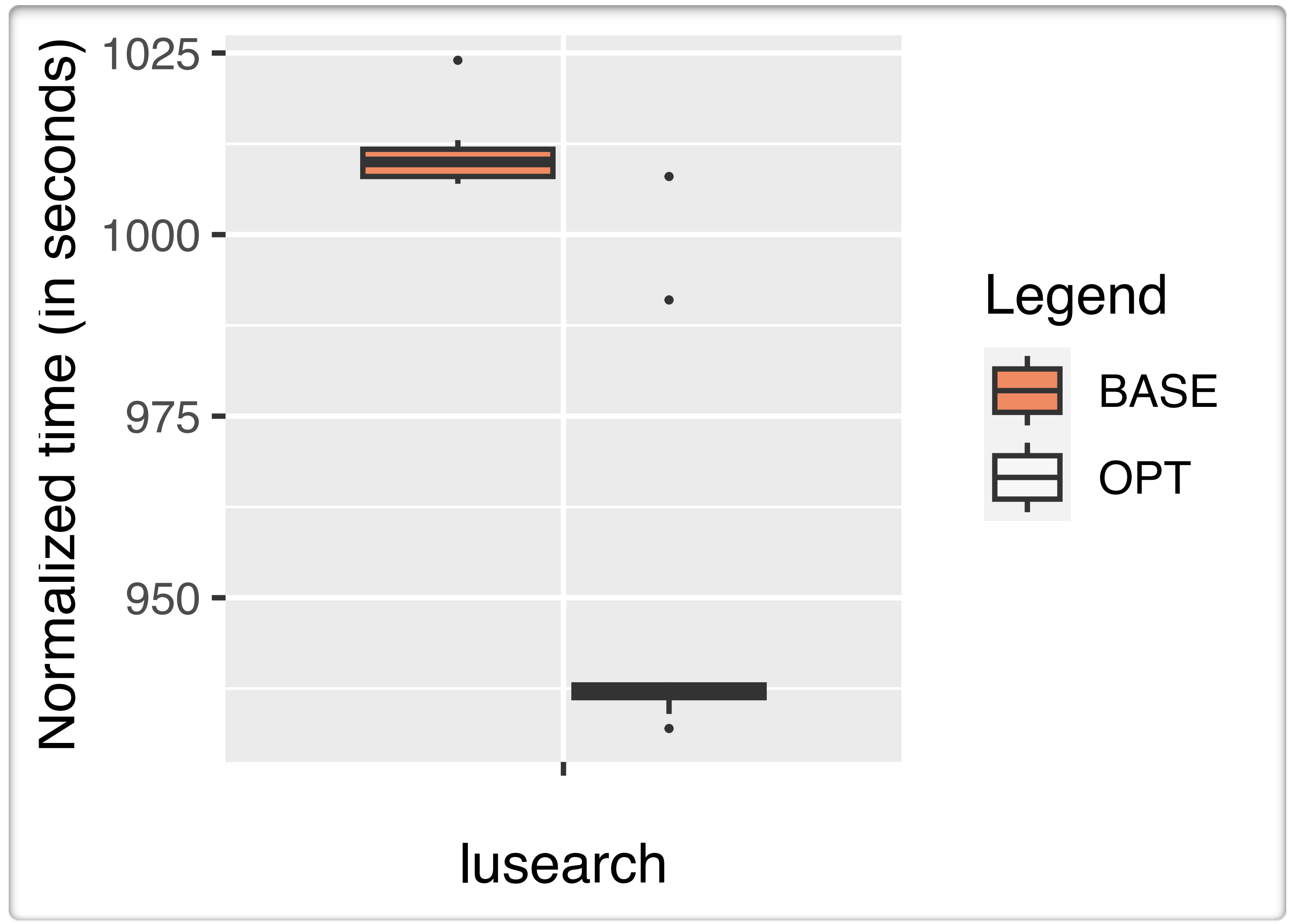
Performance

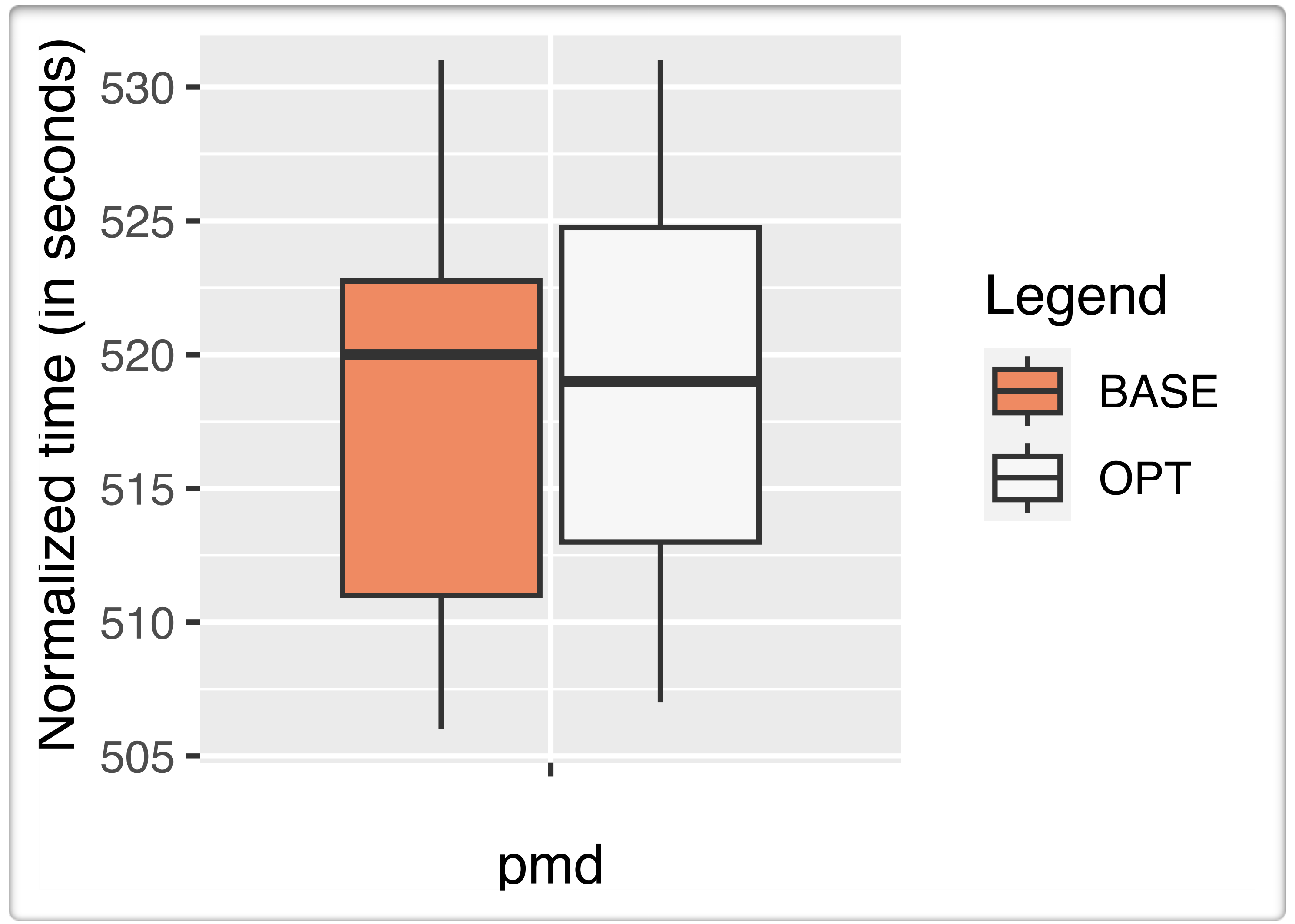




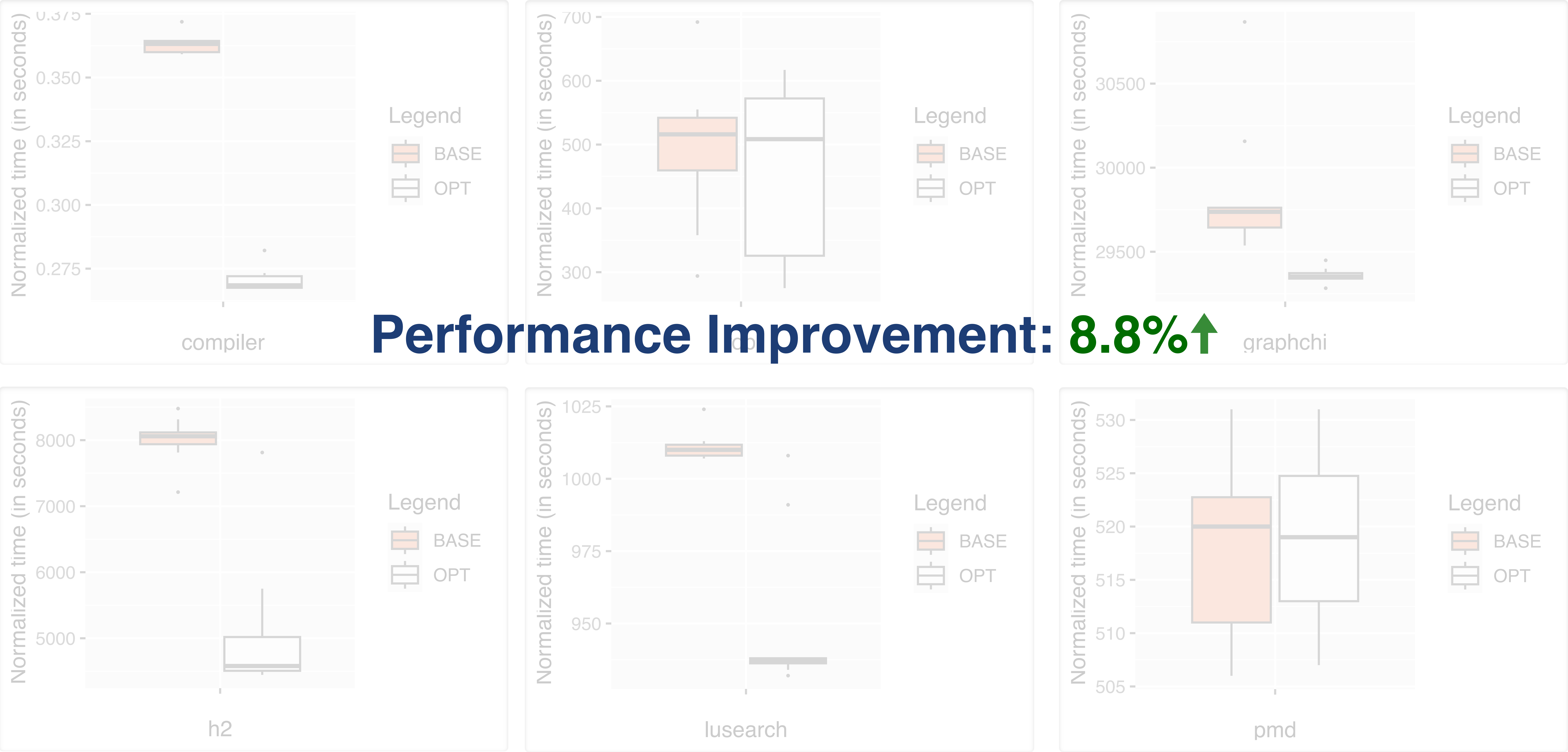






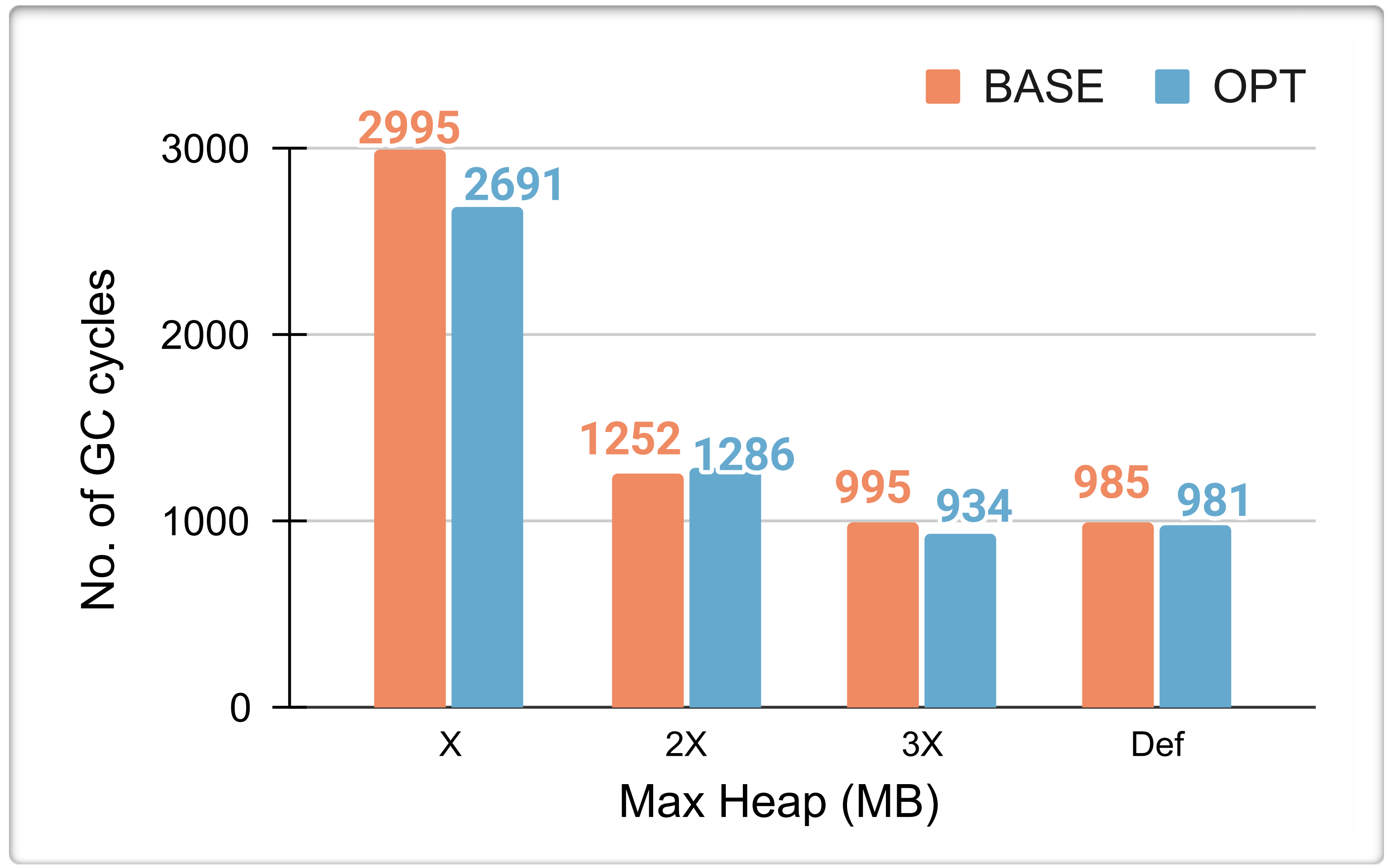


Performance

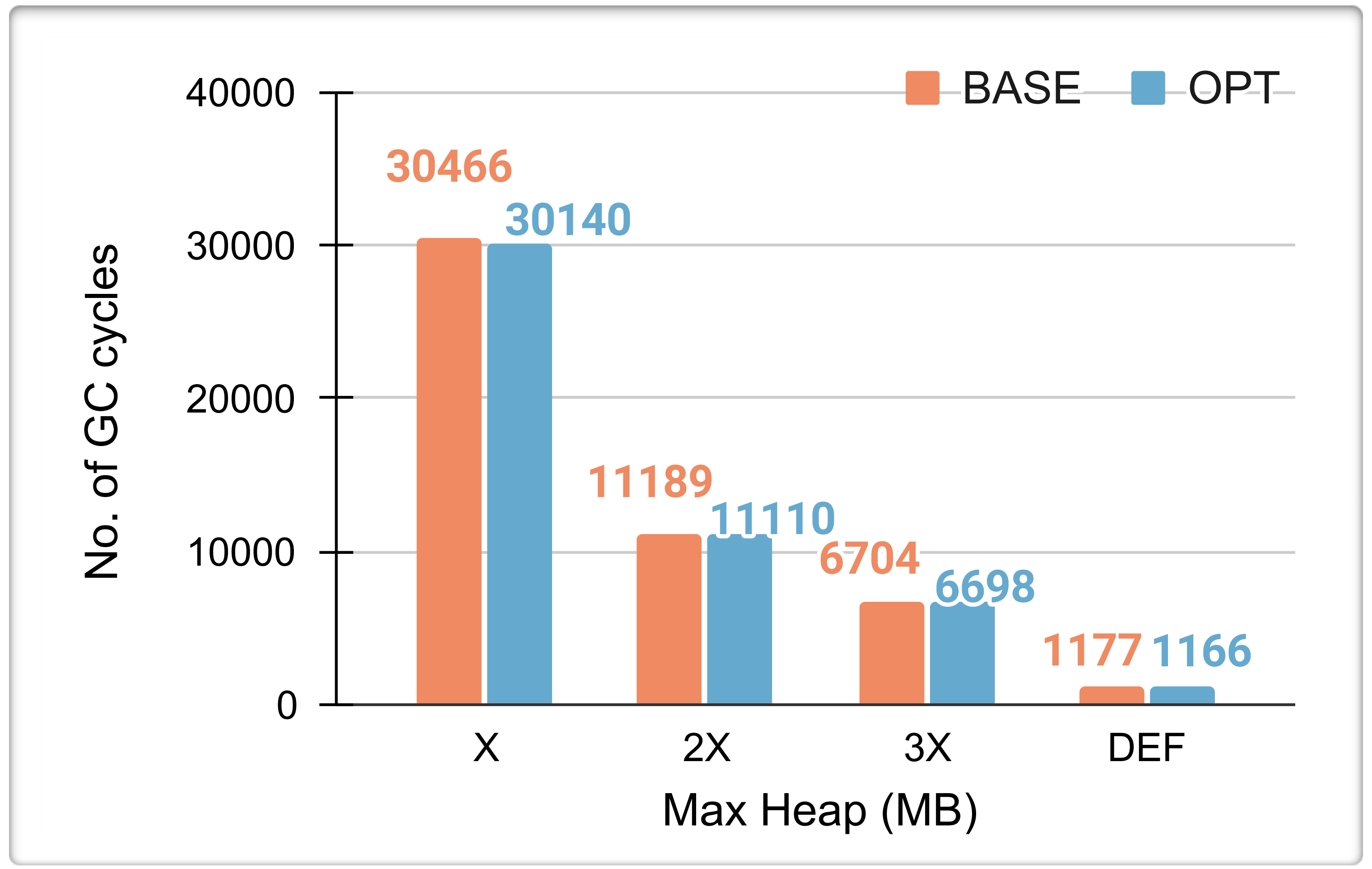


Performance Improvement: 8.8%↑

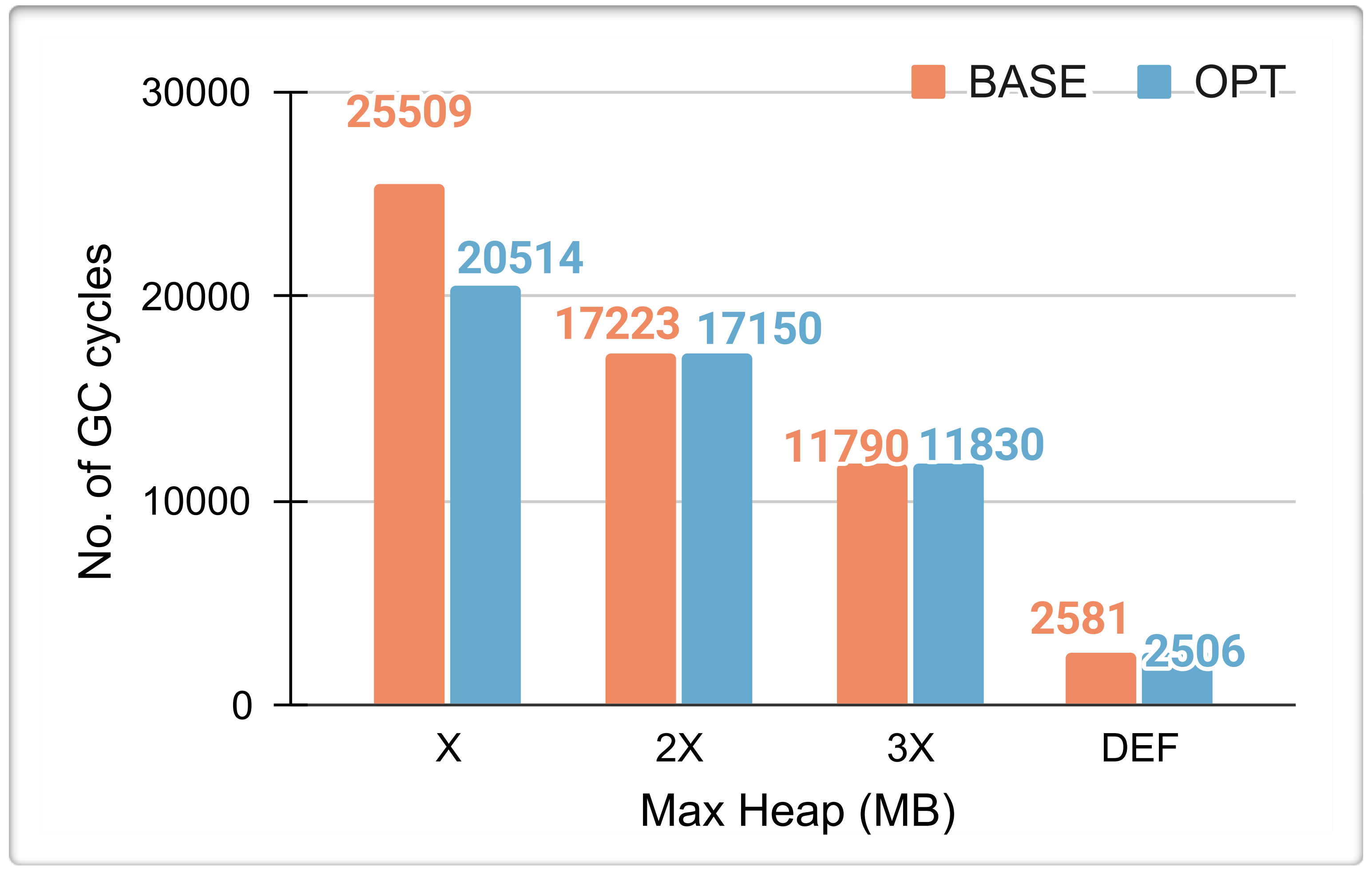
Garbage Collection



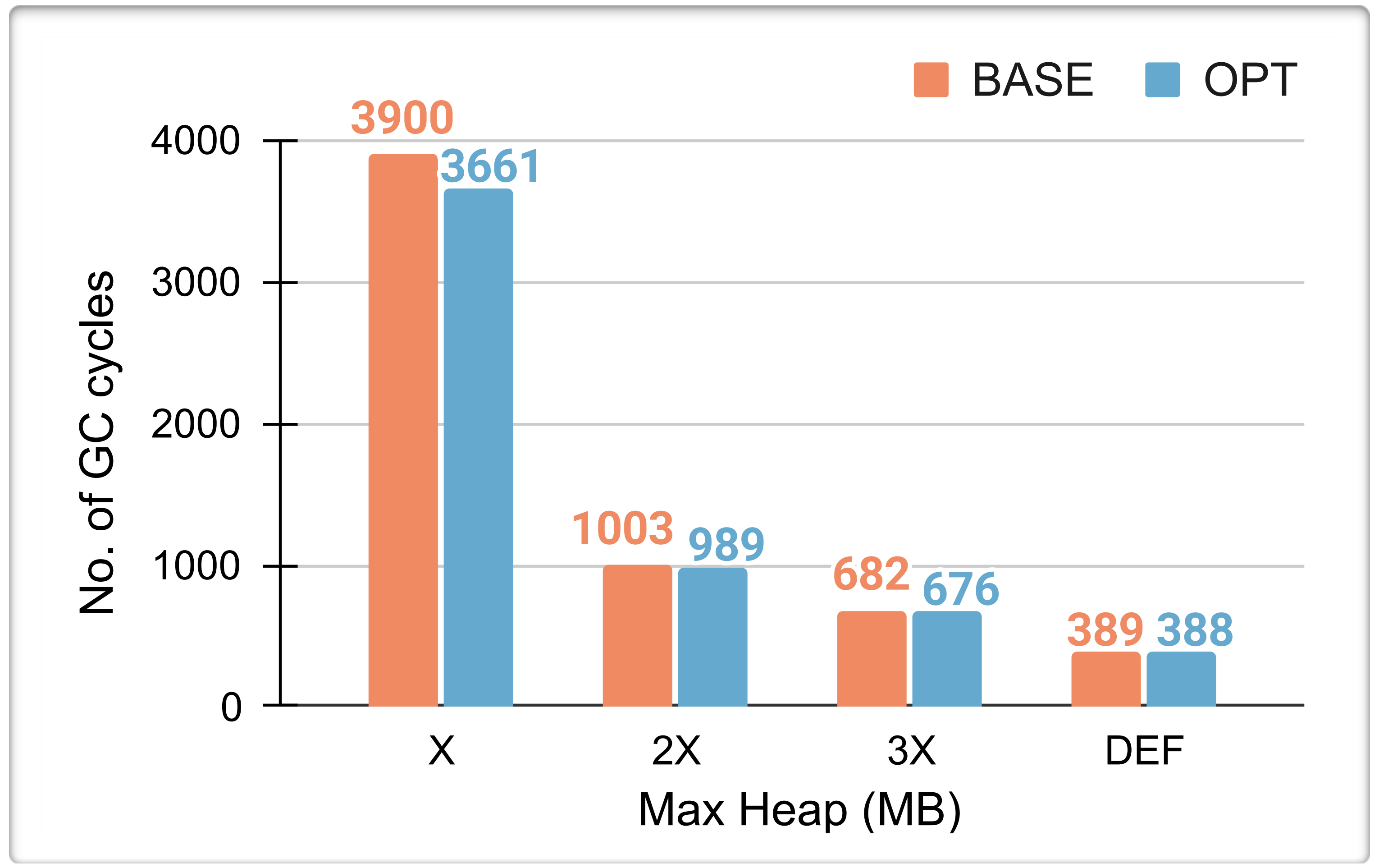
compiler



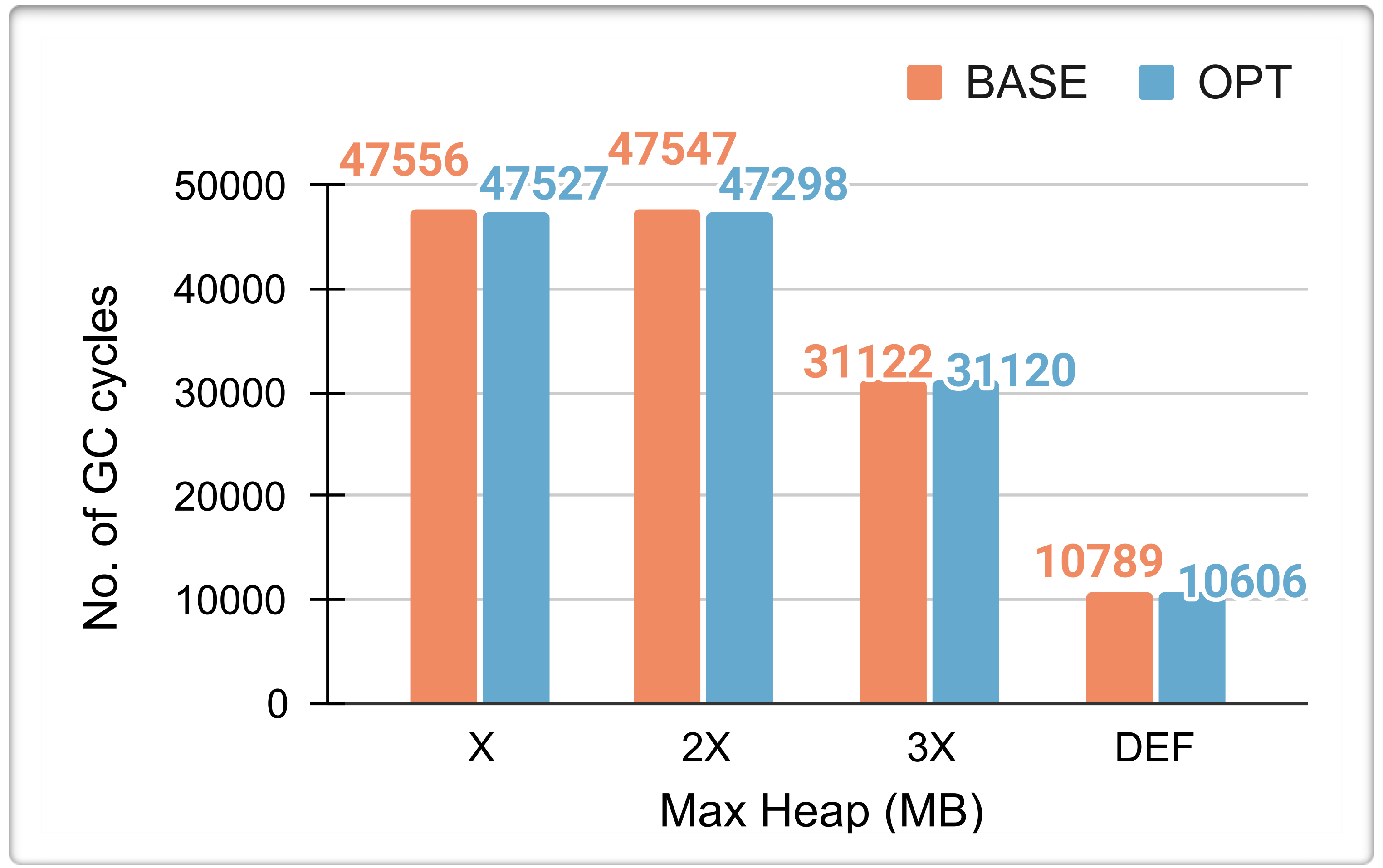
fop



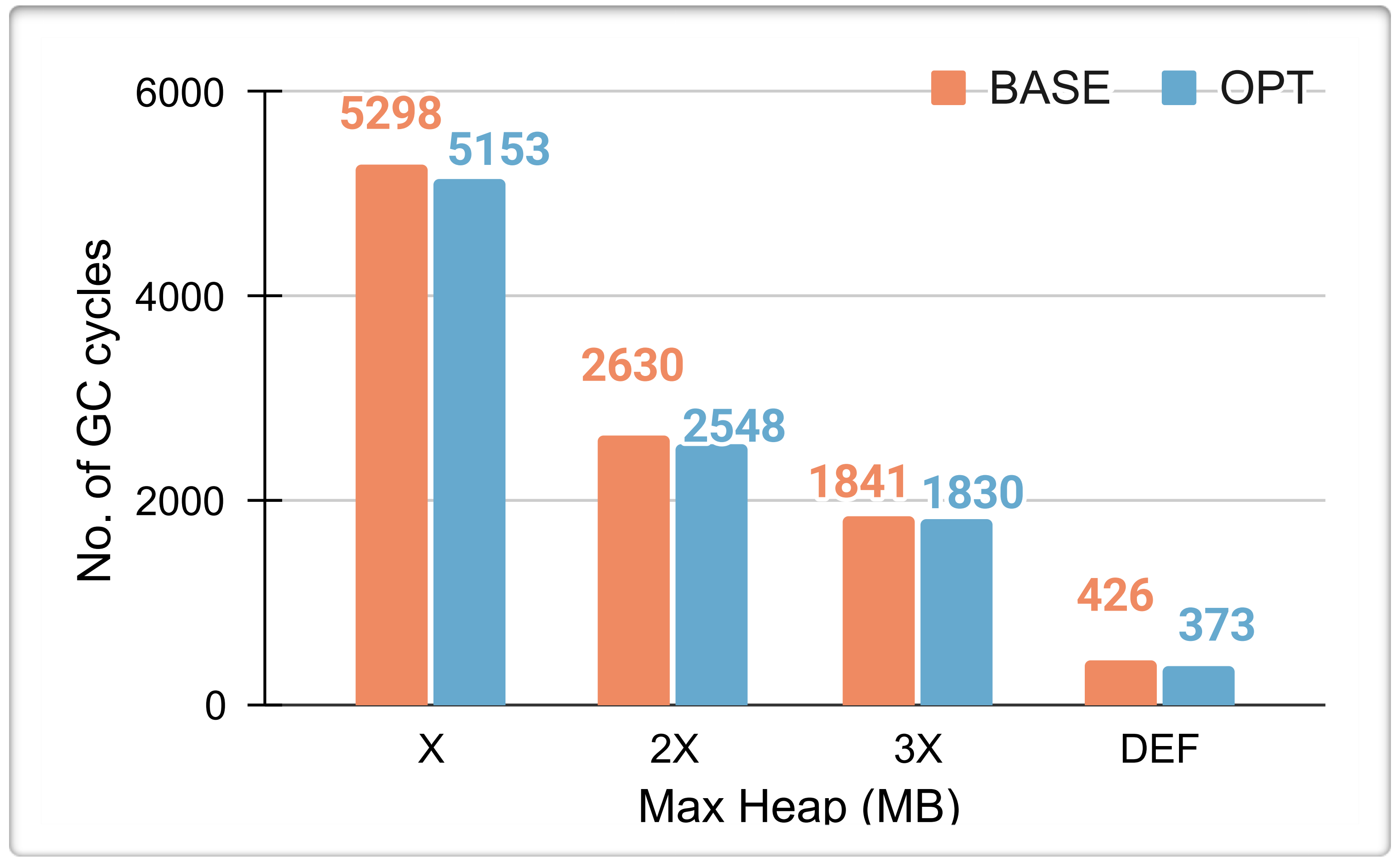
graphchi



h2

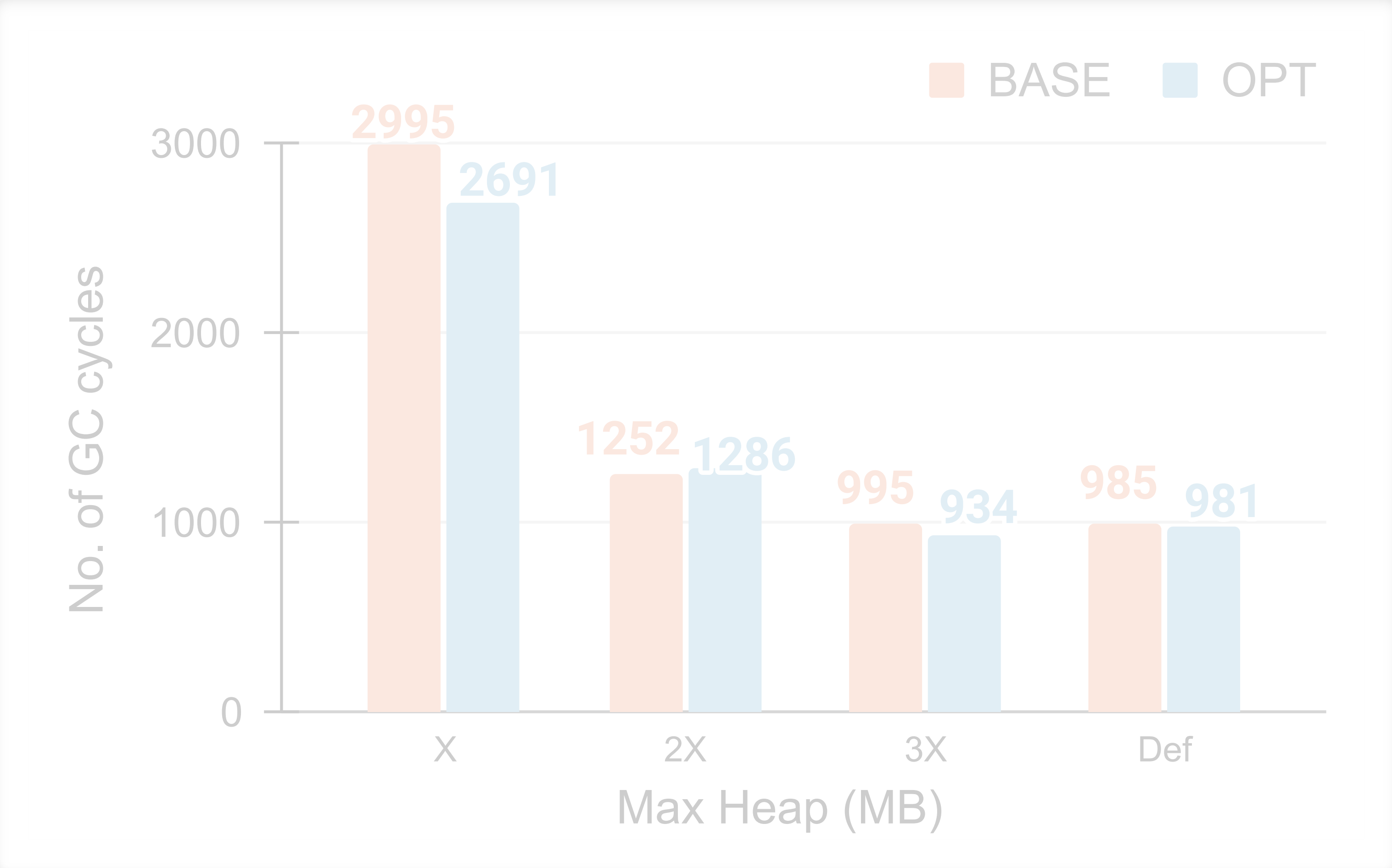


lusearch

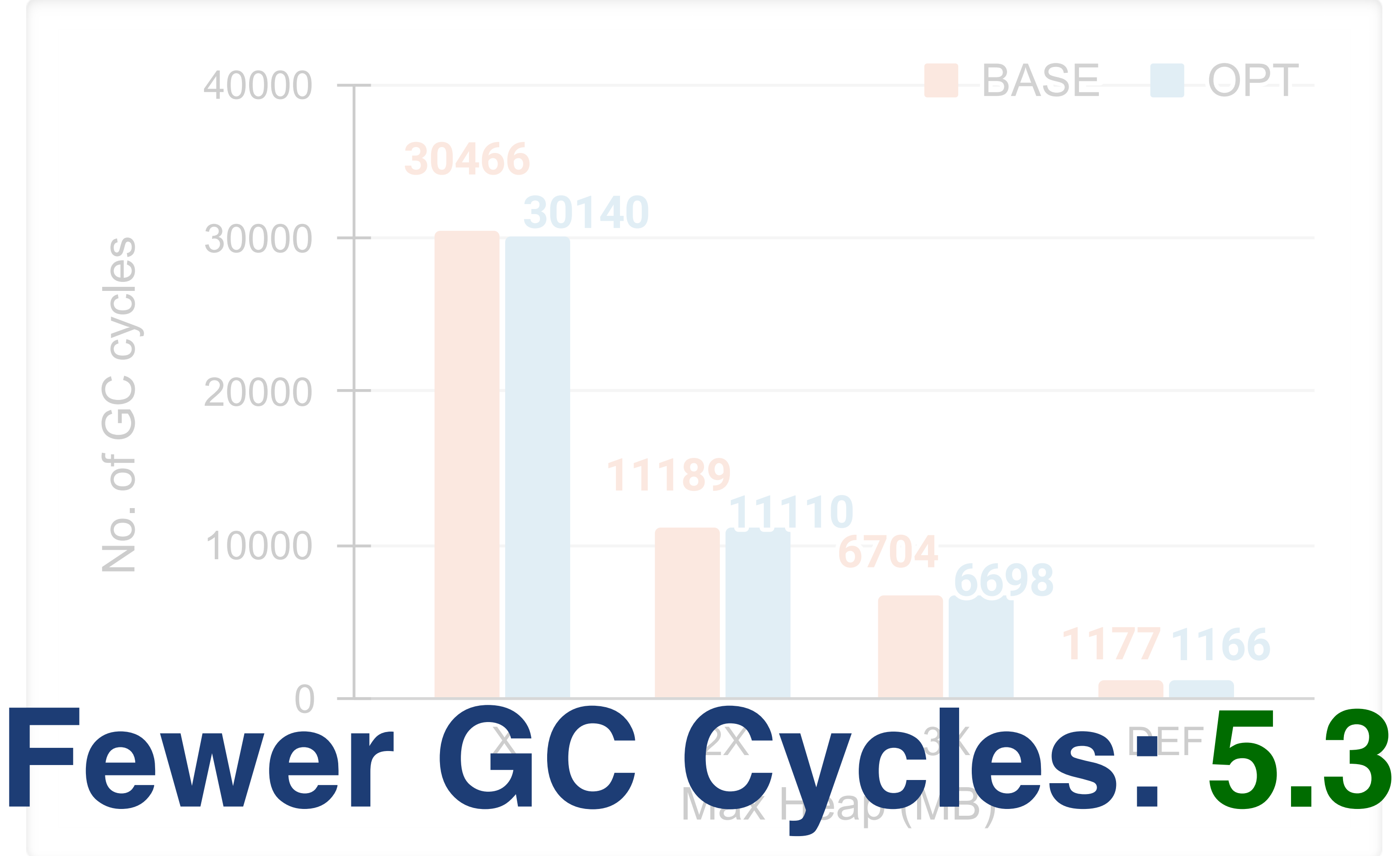


pmd

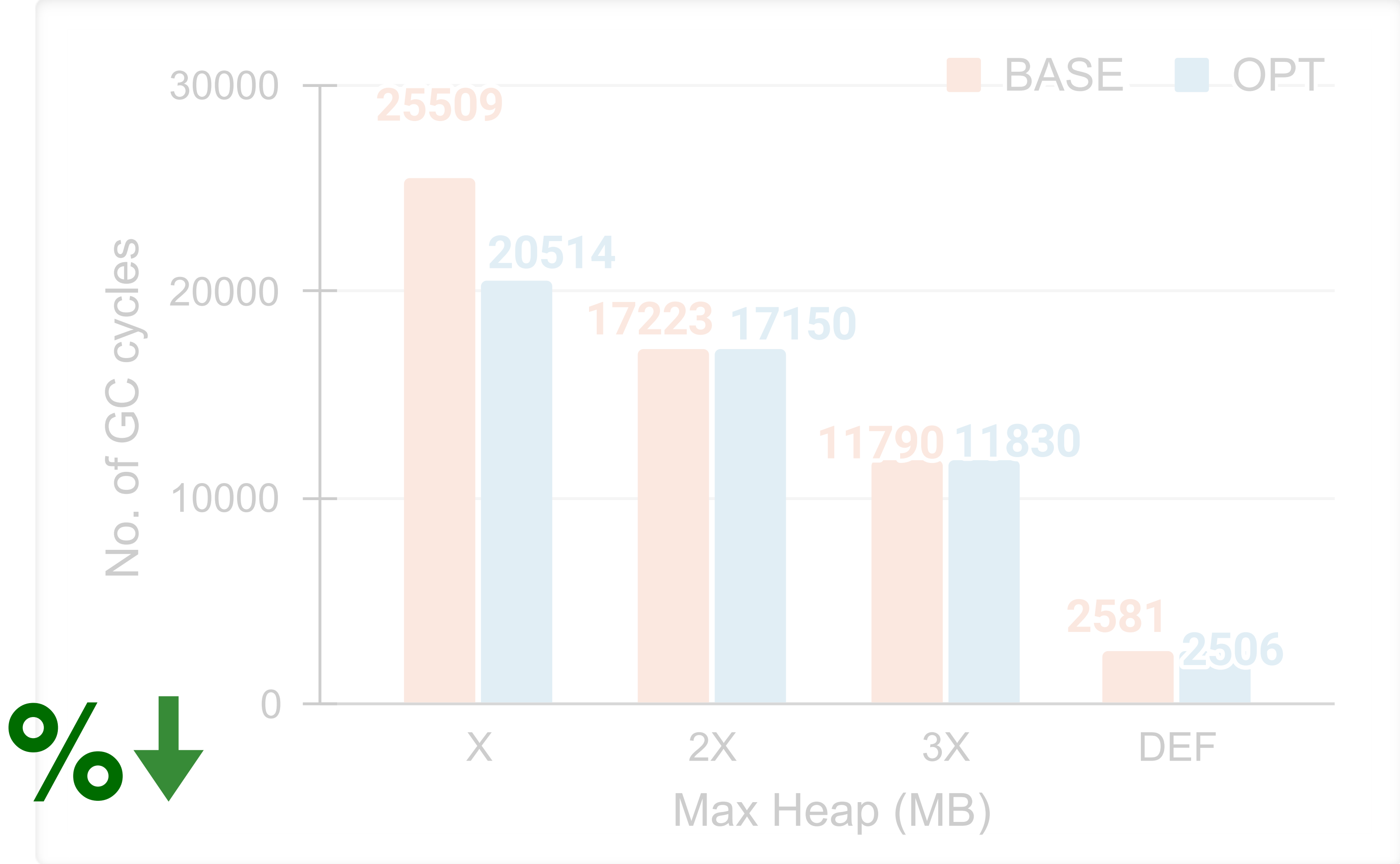
Garbage Collection



compiler

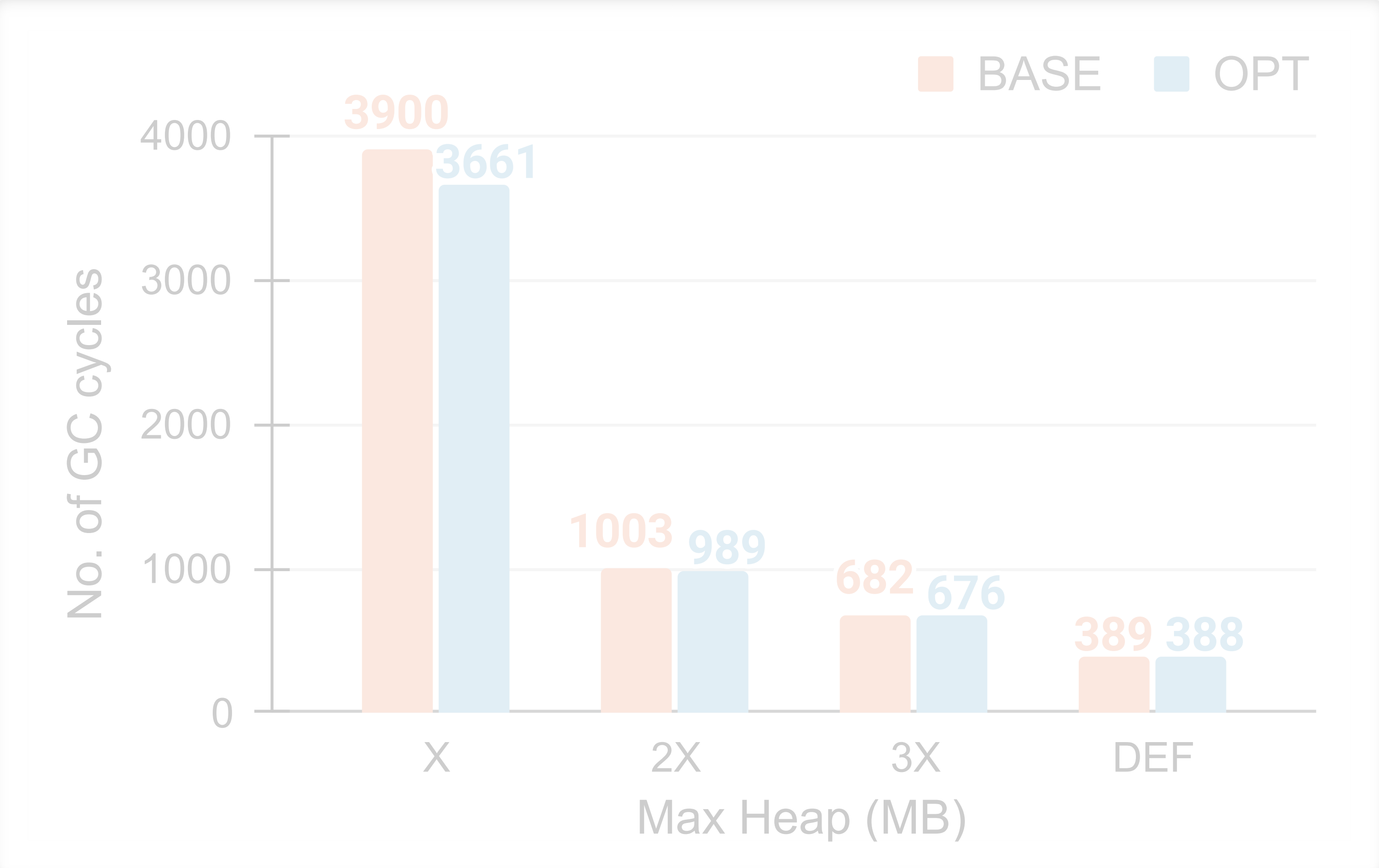


fop

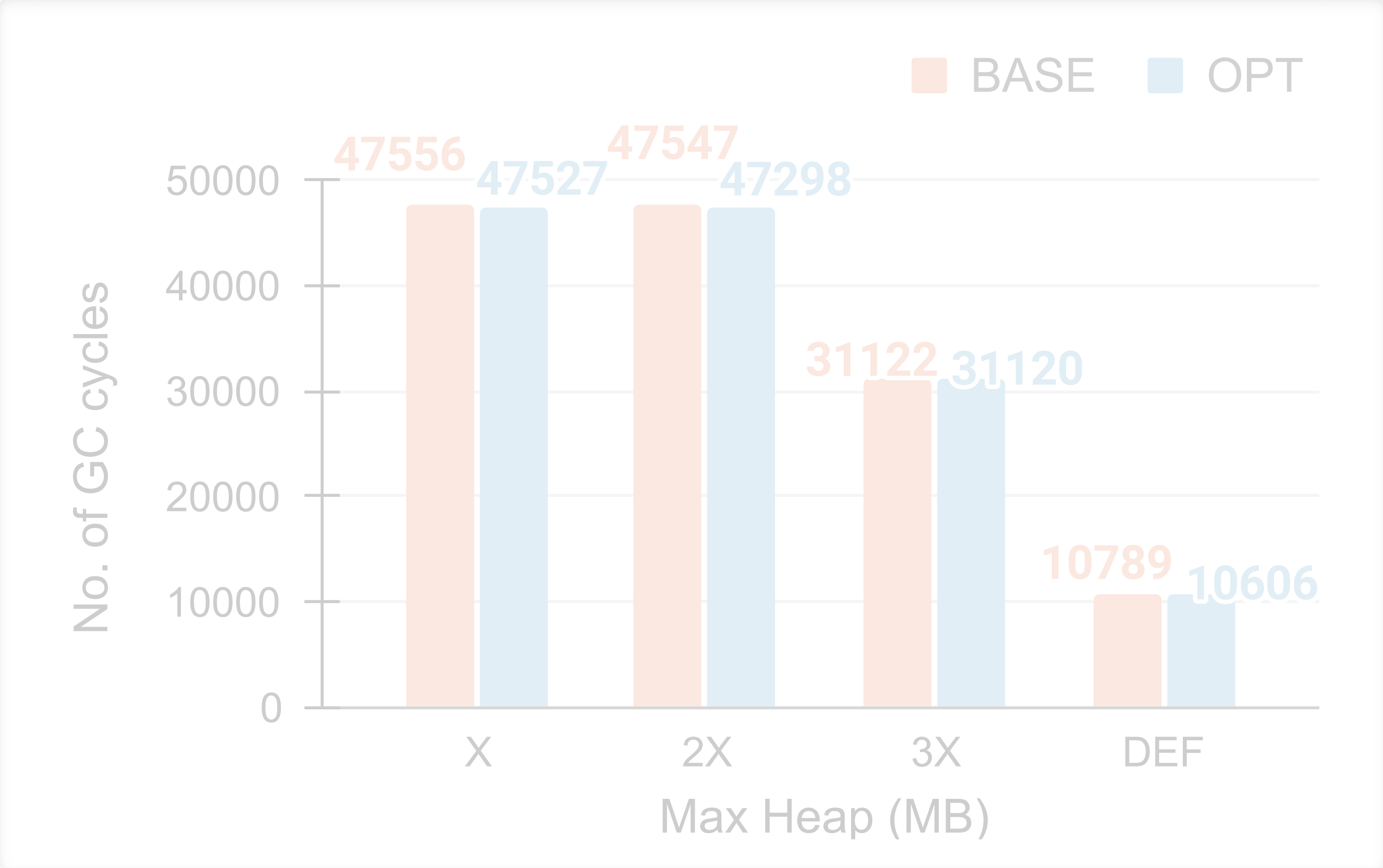


graphchi

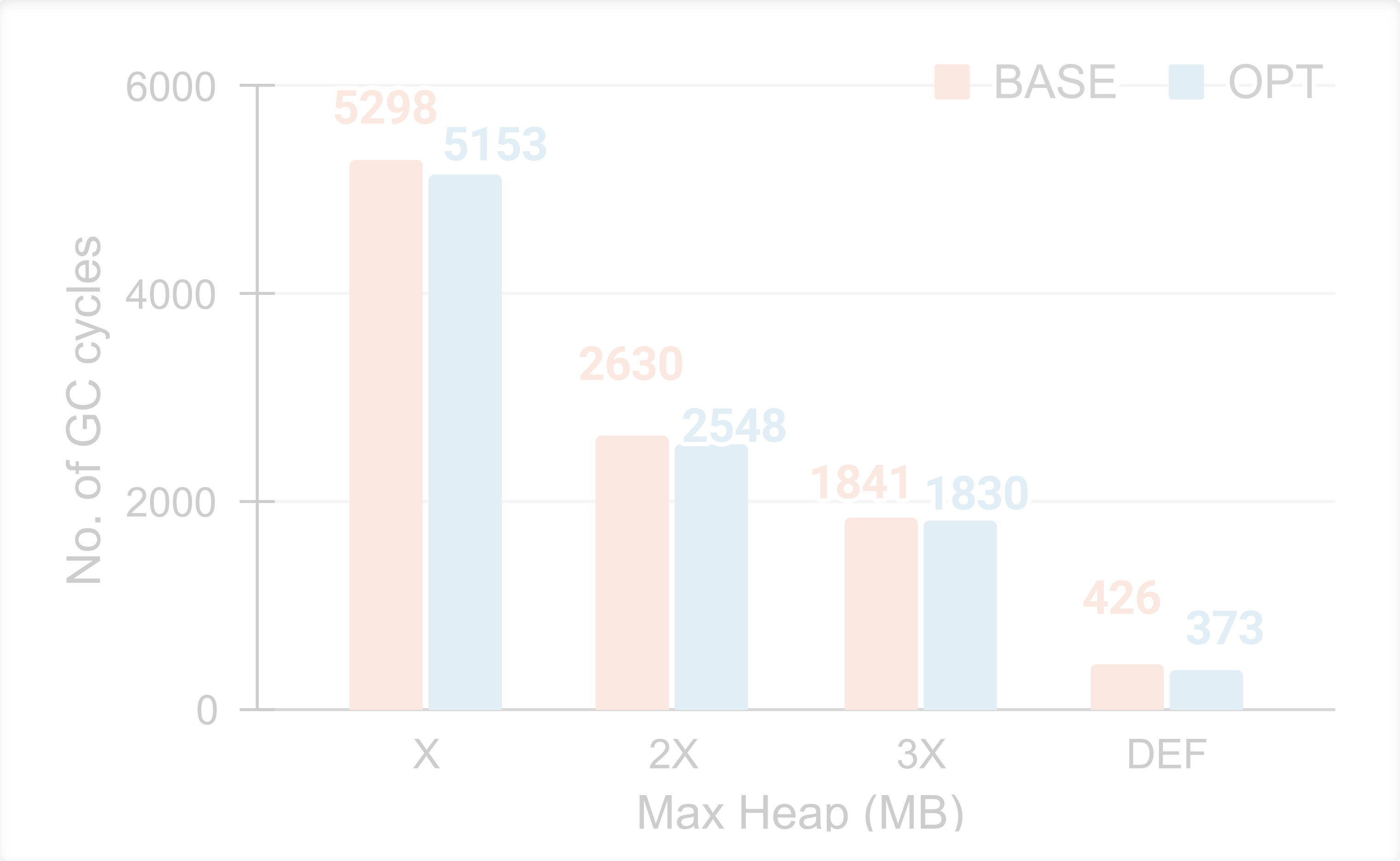
Fewer GC Cycles: 5.3%↓



h2

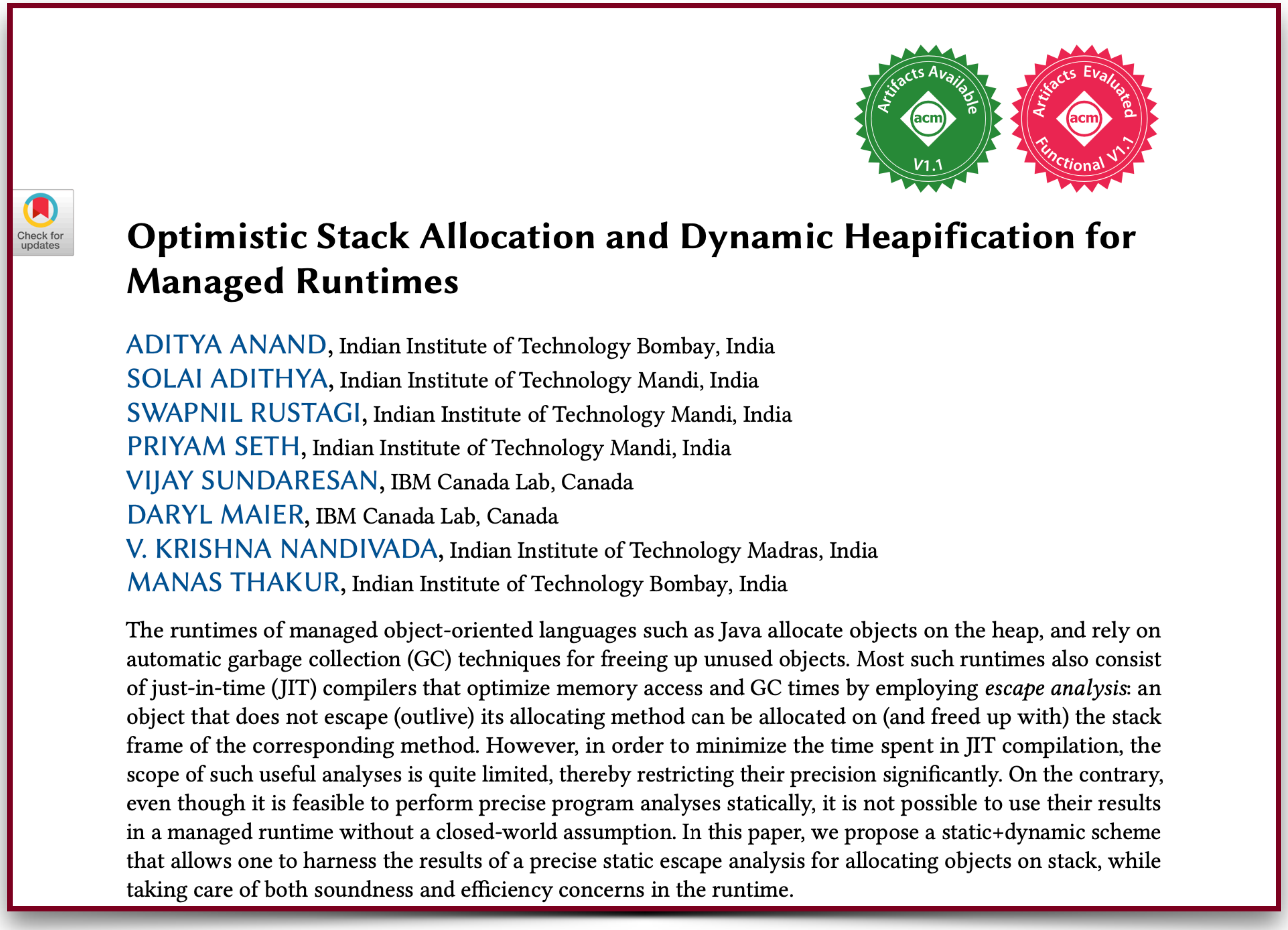


lusearch



pmd

More in Paper



Artifacts Available acm V1.1

Artifacts Evaluated acm Functional V1.1

Check for updates

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.

- Implementation of opcodes for statements that can cause an object to escape, across JIT & interpreter.
- Simulating longer runs of benchmarks with forced JIT compilation.
- Analyzing allocation sites that lead to high number of allocations.
- Cost of heapification.
- Offline cost.

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.

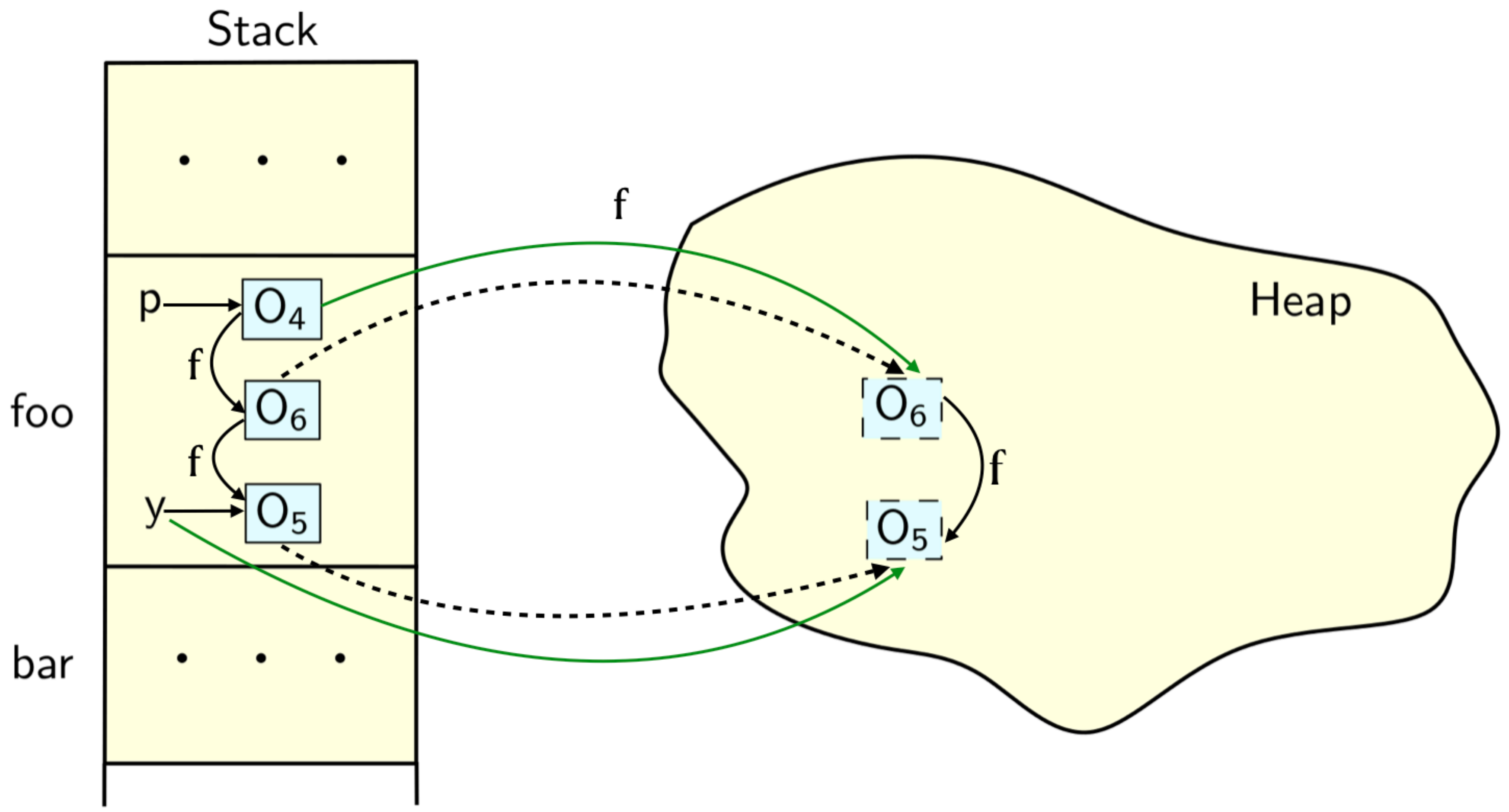
- Used static escape analysis to optimistically allocate identified objects on stack to improve the precision without thwarting the efficiency.

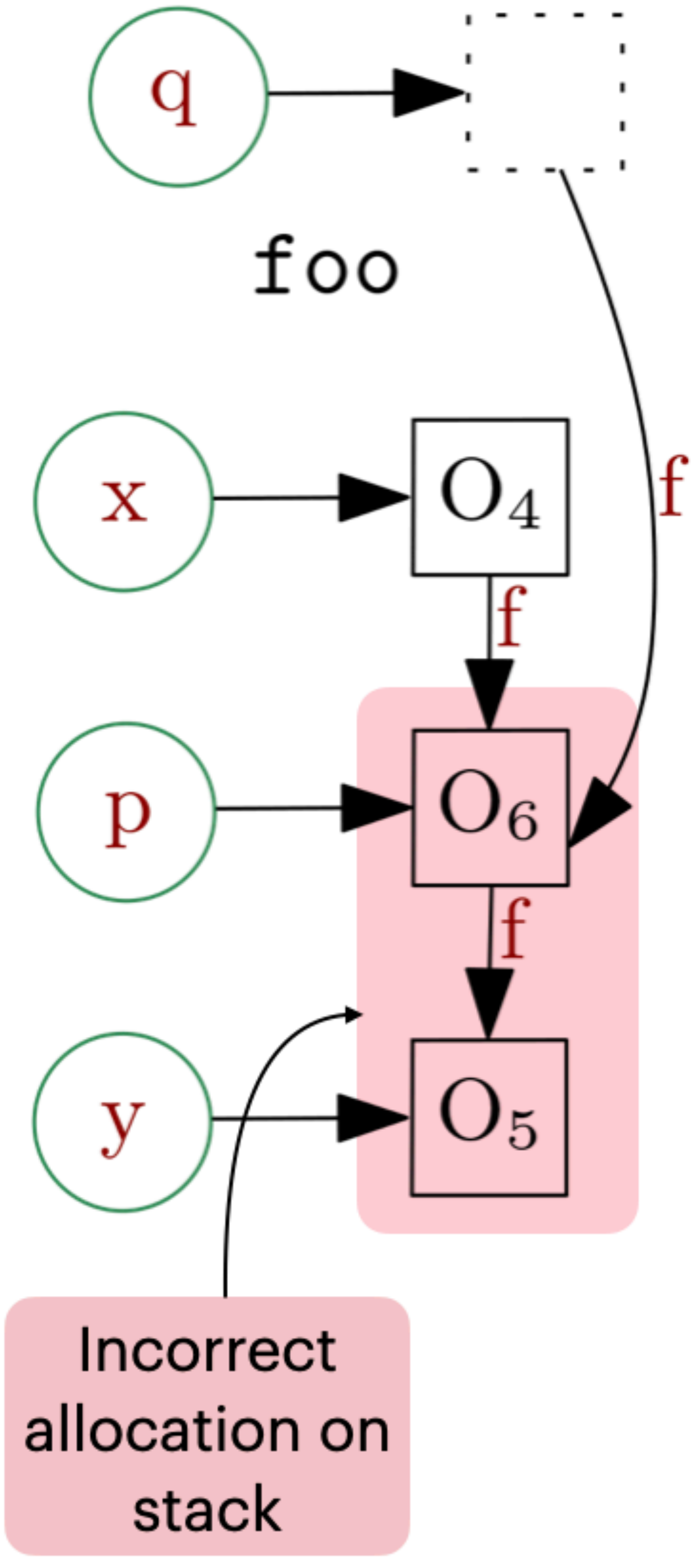
Take Aways

- 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.





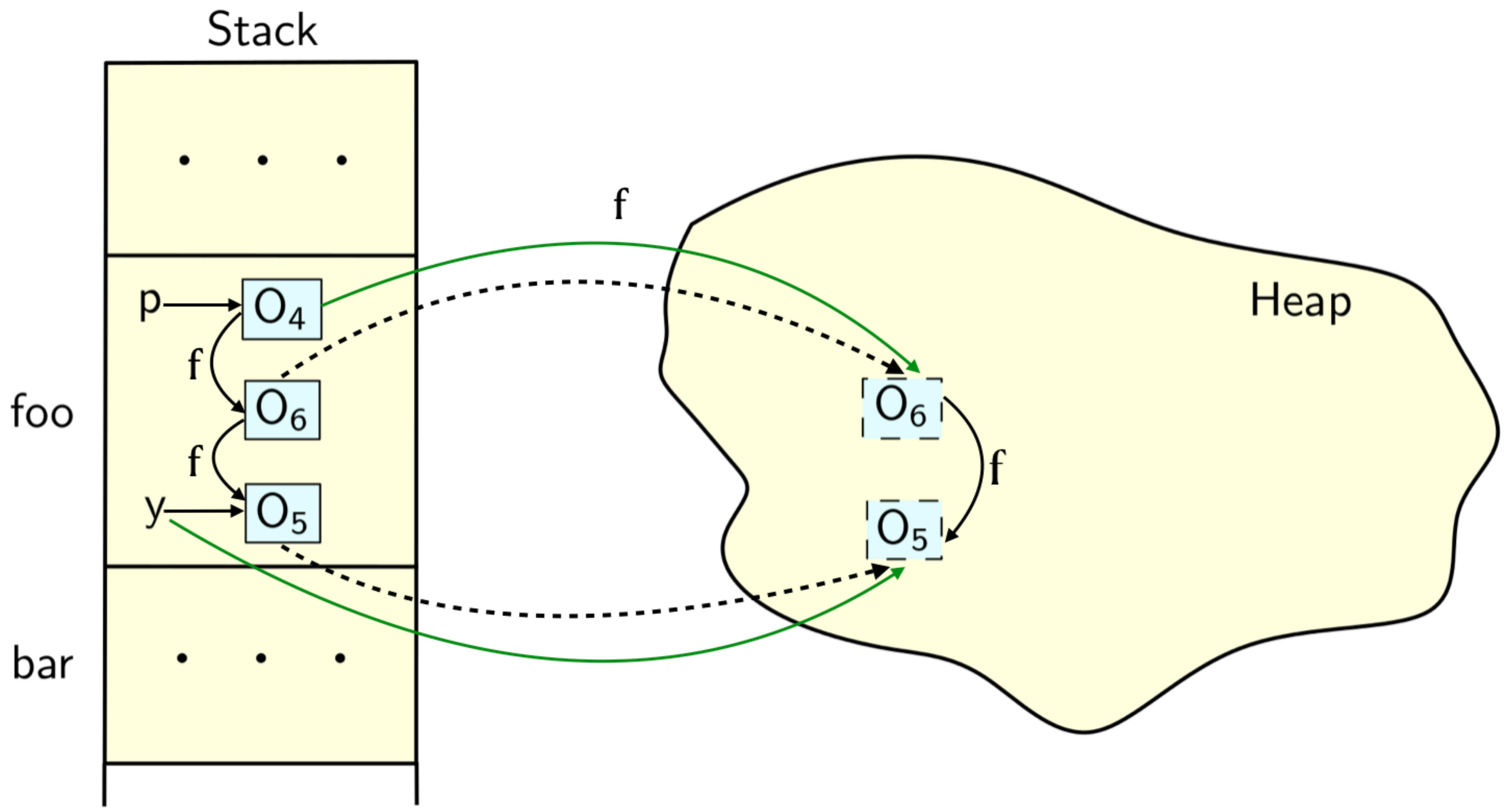
Incorrect allocation on stack

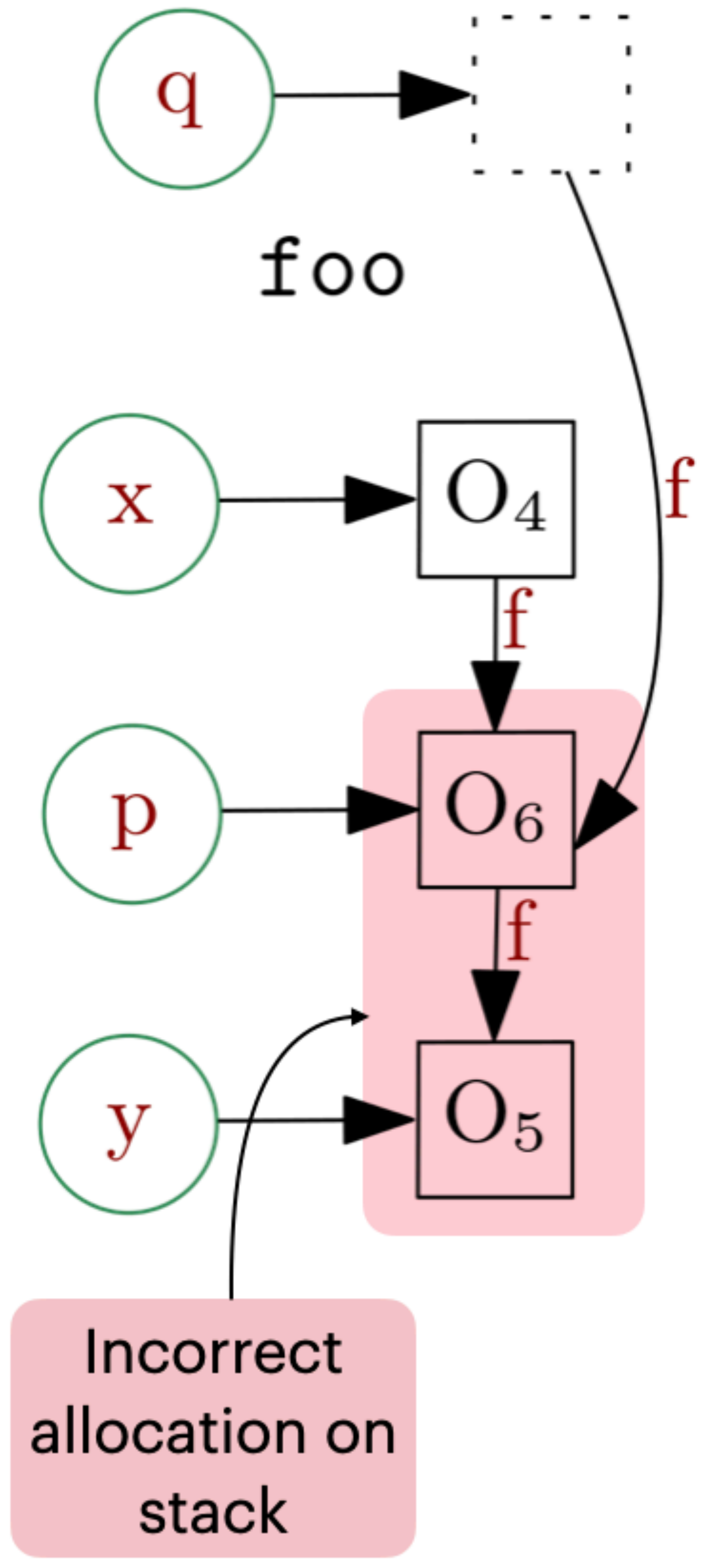
Take Aways

- 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.

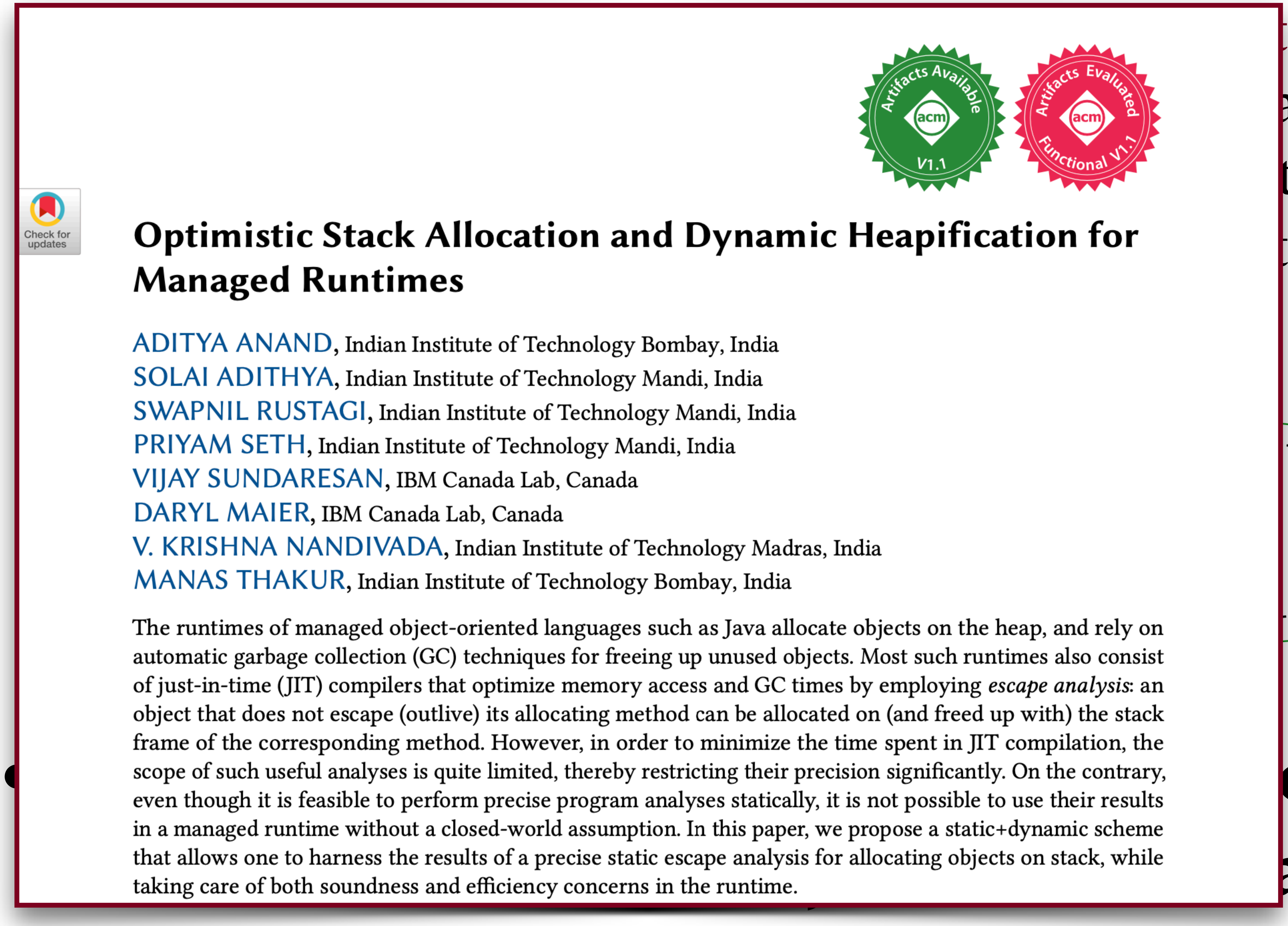




Incorrect allocation on stack

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

Take Aways

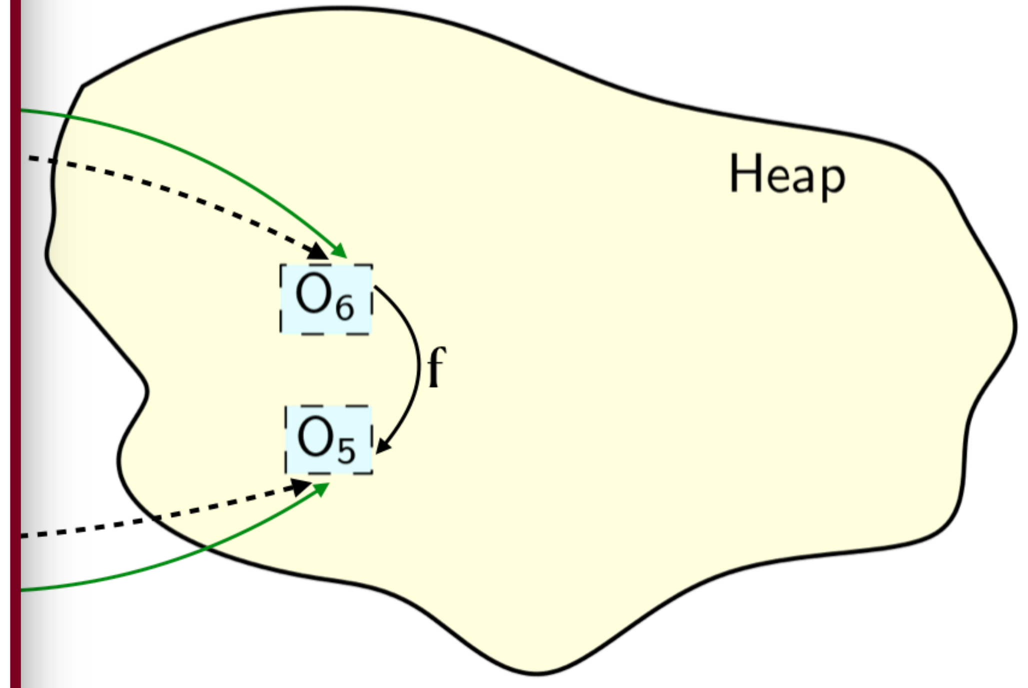


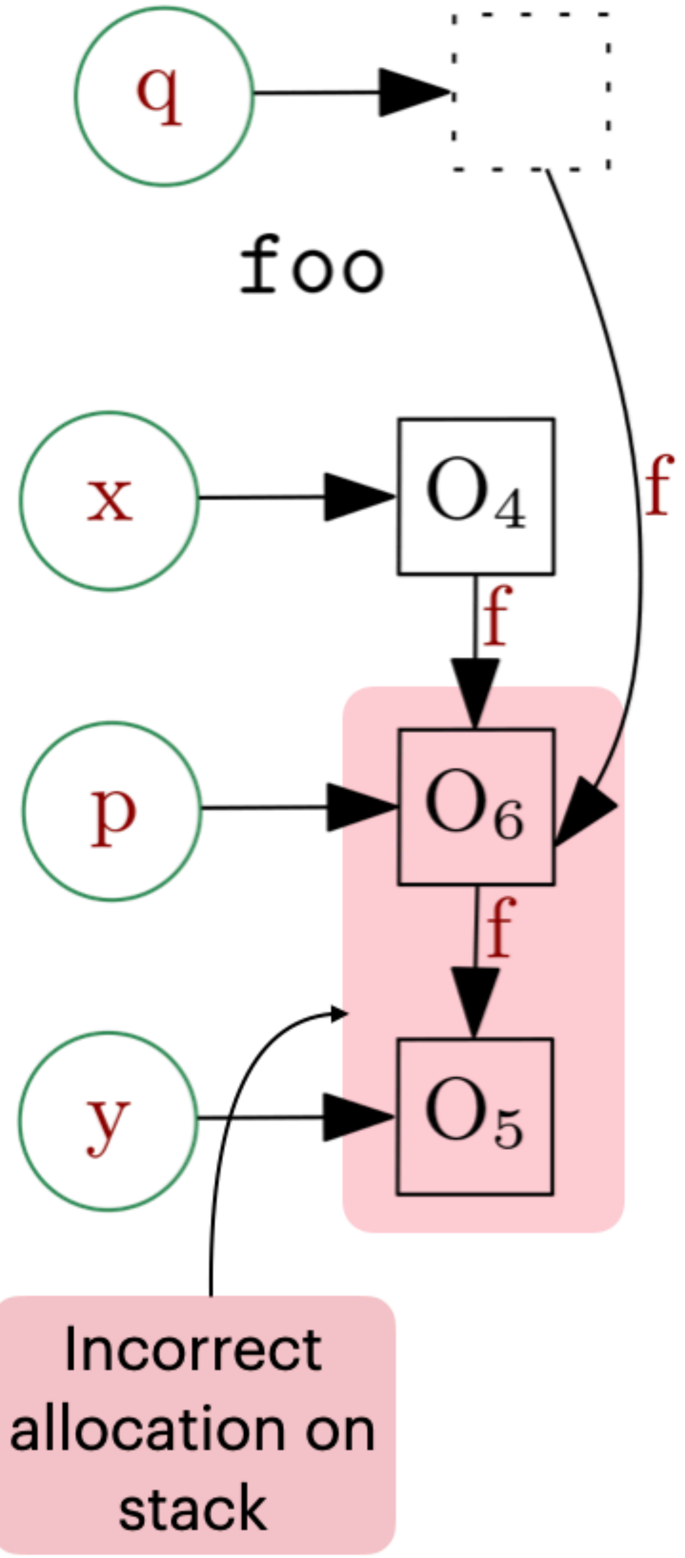
Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

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.

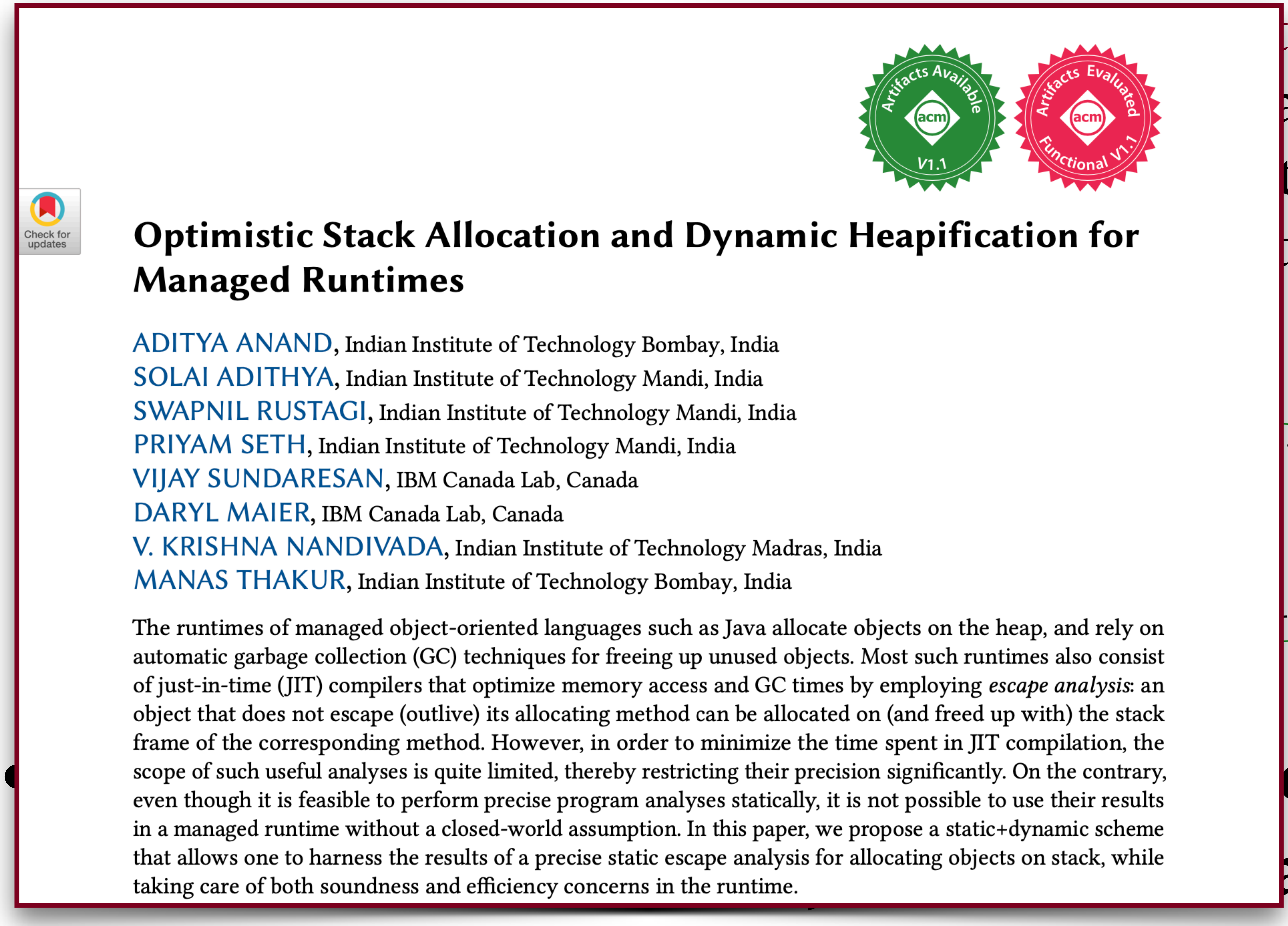
tional correctness in
analysis results do not
to the runtime
t.





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

Take Aways



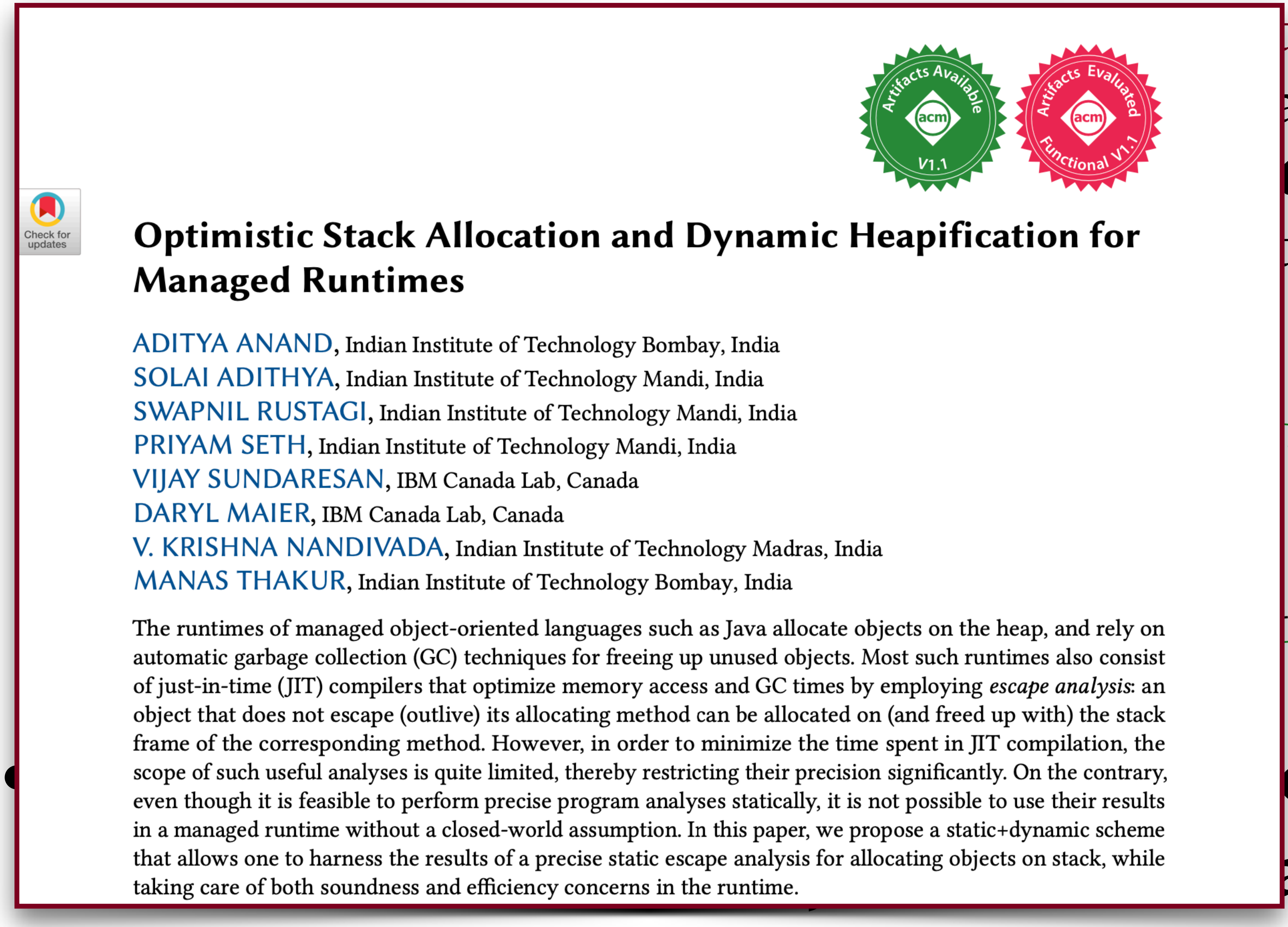
Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

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

Take Aways



Artifacts Available acm V1.1

Artifacts Evaluated acm Functional V1.1

Check for updates

Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes

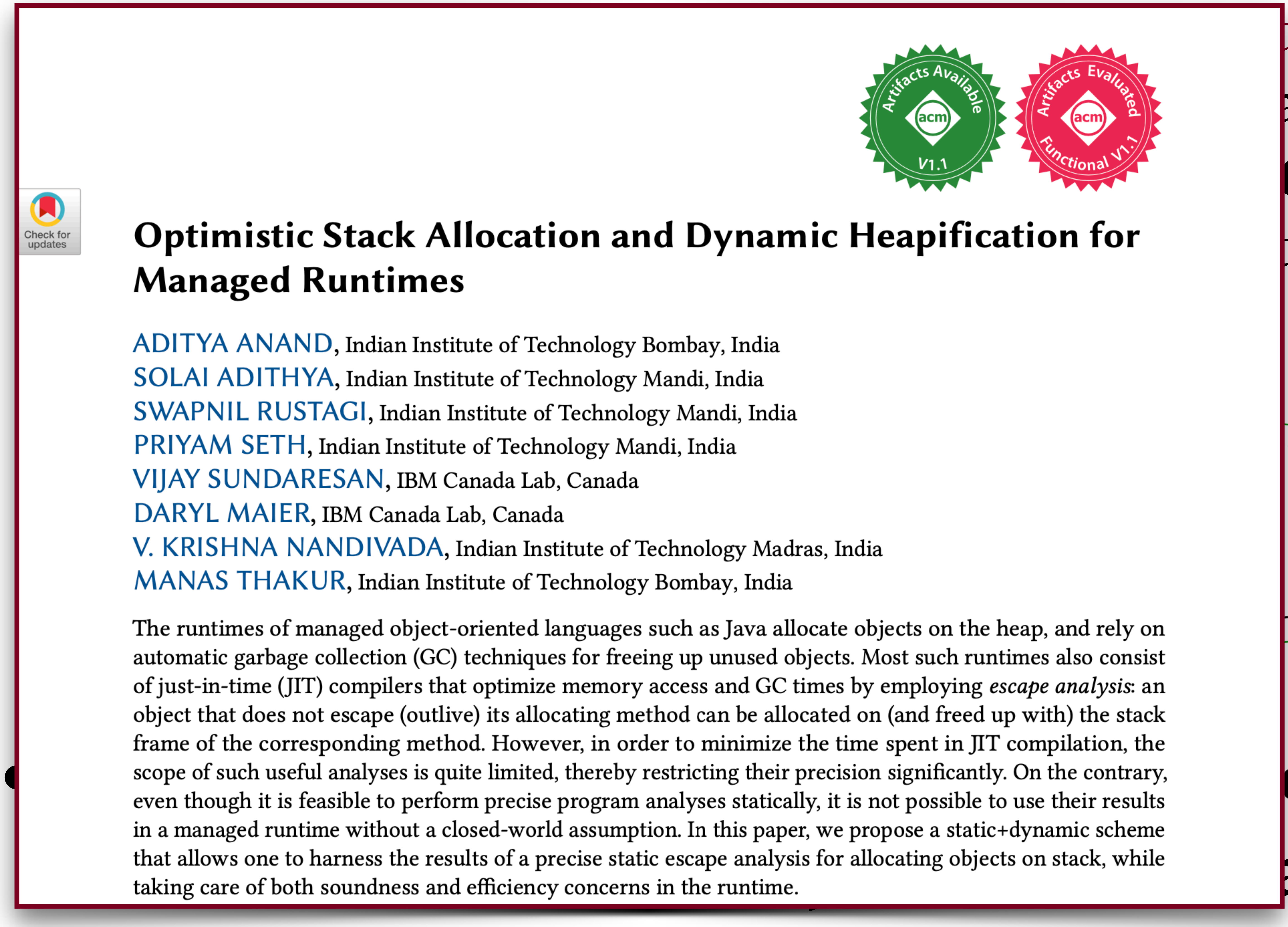
ADITYA ANAND, Indian Institute of Technology Bombay, India
SOLAI ADITHYA, Indian Institute of Technology Mandi, India
SWAPNIL RUSTAGI, Indian Institute of Technology Mandi, India
PRIYAM SETH, Indian Institute of Technology Mandi, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada 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



Artifacts Available acm V1.1

Artifacts Evaluated acm Functional V1.1

Check for updates

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

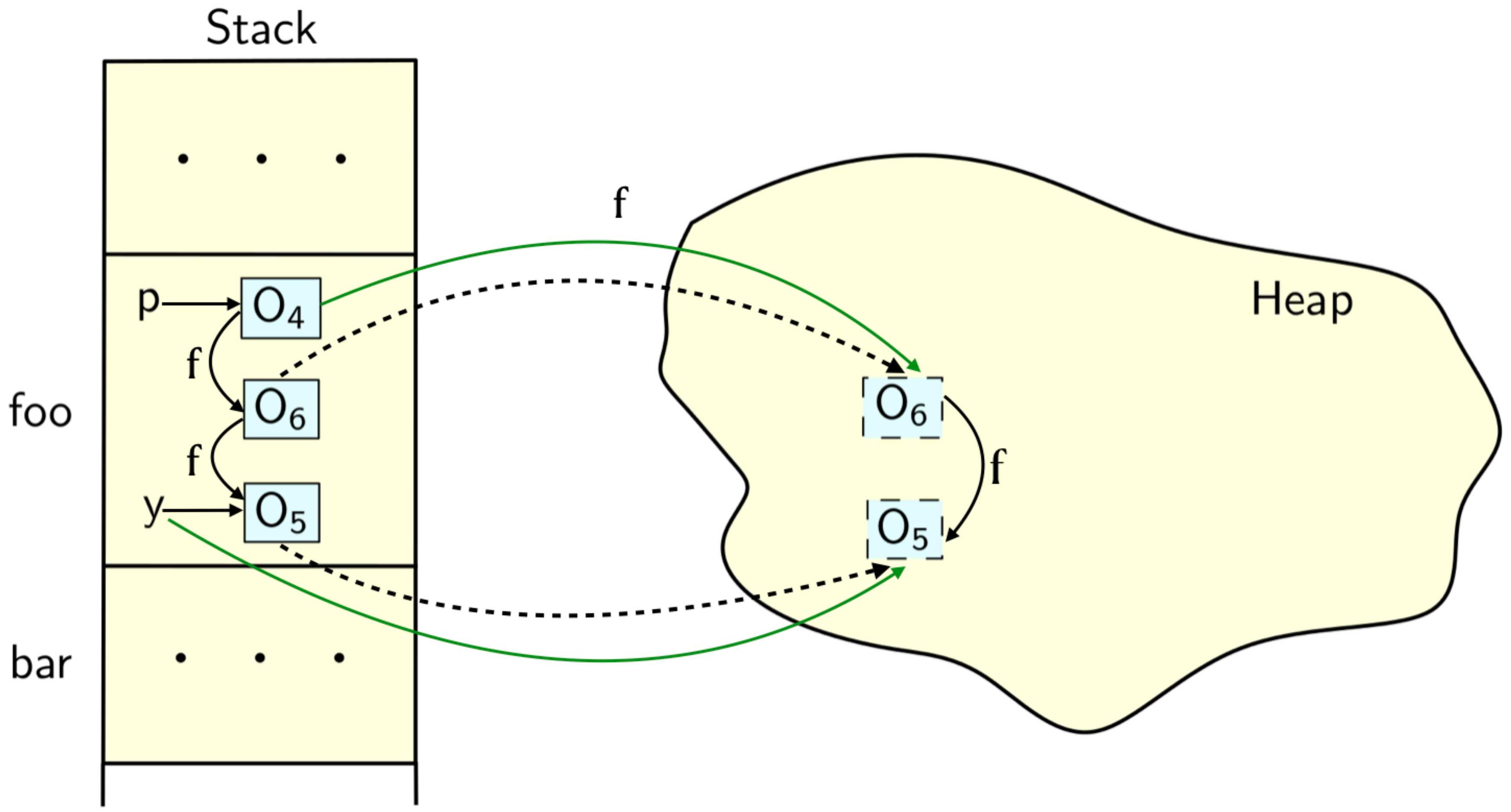
The runtimes of managed object-oriented languages such as Java allocate objects on the heap, and rely on automatic garbage collection (GC) techniques for freeing up unused objects. Most such runtimes also consist of just-in-time (JIT) compilers that optimize memory access and GC times by employing *escape 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.

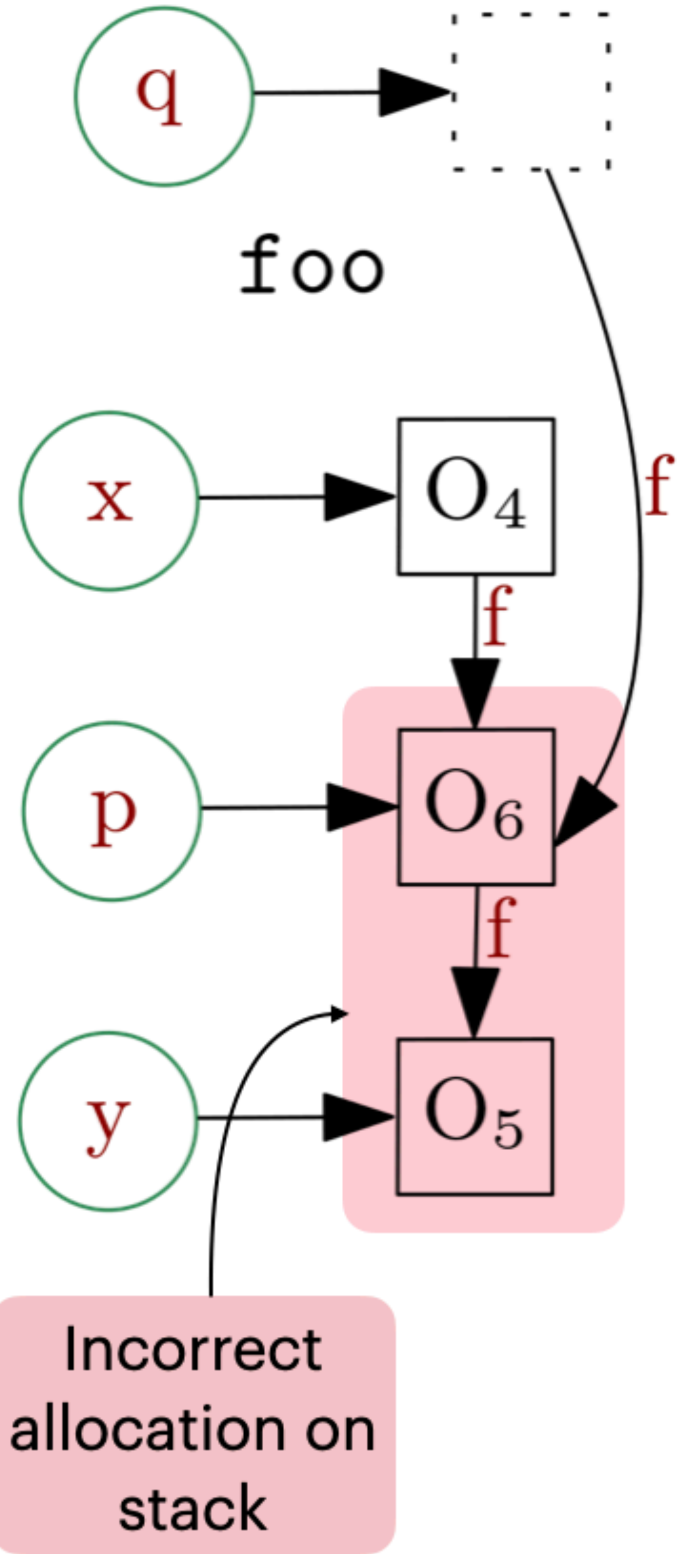
Paper Link

Thank You!! Questions?

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?

Backup

Object Representation in OpenJ9

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
  - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
  - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
    - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

- How to represent abstract objects in VM?

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
    - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

- How to represent abstract objects in VM?

- We use the Bytecodes indices (BCI's) corresponding to the bytecodes for each abstract object.

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
    - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

- How to represent abstract objects in VM?

- We use the Bytecodes indices (BCI's) corresponding to the bytecodes for each abstract object.

```
• A a = new A(); // 01                0: new #7 // class A
```

Object Representation in OpenJ9

- J9 JIT: Uses tree IL as of its analysis and transformation phases.
  - Challenge: IL is very different from the linear Java Bytecode (as well as the Java source code) available during static analysis.

- How to represent abstract objects in VM?

- We use the Bytecodes indices (BCI's) corresponding to the bytecodes for each abstract object.

```
• A a = new A(); // 01                0: new #7 // class A
```

```
• Main.foo() V [0]
```

Related Work

- C2 just-in-time (JIT) compiler of the HotSpot VM uses escape analysis to decompose objects into scalar variables on the stack.
  - It uses connection graphs (which do not maintain points-to relationships directly but allow one to perform reachability checks faster) to perform synchronization elision and scalar replacement.
- GraalVM uses a partial-escape analysis to enable scalar replacement in parts of a program when it cannot be performed throughout the program.
  - However stack allocation is possible in many scenarios where scalar replacement is not.
- GraalVM also uses escape analysis results that works in presence of dynamic classloading for the C1 compiler. It reallocates objects replaced by scalars if the VM deoptimizes to the interpreter.