

软件分析与验证前沿

苏亭

软件科学与技术系

Pointer Analysis

Introducing Pointers

Example without pointers

```
[x == 1] → x = 1;  
         y = x;  
[y == 1] → assert(y == 1)
```

Same example with pointers

```
x = new Circle();  
x.radius = 1;  
y = x.radius;  
assert(y == 1)
```

Introducing Pointers

Example without pointers

```
[x == 1] → x = 1;  
[y == 1] → y = x;  
[y == 1] → assert(y == 1)
```

Same example with pointers

```
x = new Circle();  
x.radius = 1;  
[x.radius == 1] → y = x.radius;  
[y == 1] → assert(y == 1)
```

Pointer Aliasing

- Situation in which same address referred to in different ways

	<pre>x = new Circle();</pre>		<pre>Circle x = new Circle();</pre>
	<pre>x.radius = 1;</pre>		<pre>Circle z = ?</pre>
<pre>[x.radius == 1]</pre>	<pre>→ y = x.radius;</pre>	<pre>[x.radius == 1]</pre>	<pre>→ x.radius = 1;</pre>
<pre>[y == 1]</pre>	<pre>→ assert(y == 1)</pre>	<pre>[x.radius == ?]</pre>	<pre>→ z.radius = 2;</pre>
			<pre>y = x.radius;</pre>
			<pre>assert(y == 1)</pre>

(May- or Must-) Alias Analysis

- Aliasing occurs when two variables refer to *the same* memory location. Aliasing occurs in languages with *reference parameters*, *pointers*, or *arrays*.
- Let *a* and *b* be references to memory locations. At a program point *p*:
 - *may-alias(a, b)* such that there exists *at least one execution path* to *p*, where *a* and *b* refer to the same memory location.
 - *must-alias(a, b)* such that on *all execution paths* to *p*, *a* and *b* refer to the same memory location.

May-Alias Analysis

x MAY-ALIAS z?

[x != z] → Circle x = new Circle();
[x.radius == 1, x != z] → Circle z = new Circle();
[x.radius == 1] → x.radius = 1;
[y == 1] → z.radius = 2;
→ y = x.radius;
→ assert(y == 1)

May-Alias Analysis == Pointer Analysis

Must-Alias Analysis

x MUST-ALIAS z?

```
Circle x = new Circle();  
Circle z = x;  
x.radius = 1;  
z.radius = 2;  
y = x.radius;  
assert(y == 1) y == 2
```

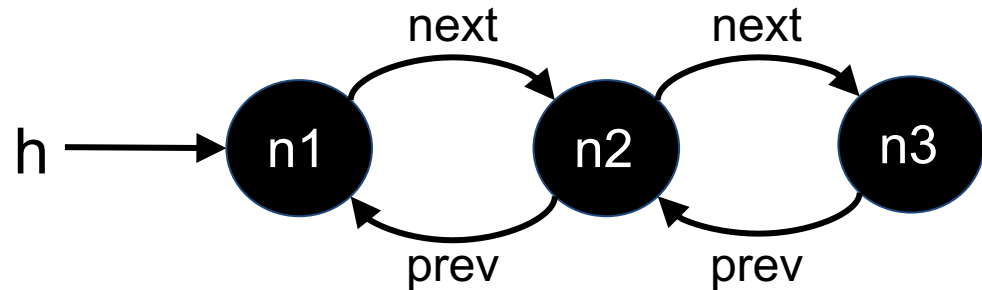
[x == z] →
[x.radius == 1, x == z] →
[x.radius == 2] →
[y == 2] →

- May-Alias and Must-Alias are dual problems
- Must-Alias more advanced, less useful in practice
- Focus of this Lesson: May-Alias Analysis

Why Is Pointer Analysis Hard?

```
class Node {  
    int data;  
    Node next, prev;  
}
```

```
Node h = null;  
for (...) {  
    Node v = new Node();  
    if (h != null) {  
        v.next = h;  
        h.prev = v;  
    }  
    h = v;  
}
```



h.data
h.next.prev.data
h.next.next.prev.prev.data
h.next.prev.next.prev.data

And many more ...

Approximation to the Rescue

- Pointer analysis problem is undecidable

=> We must sacrifice some combination of:
Soundness, Completeness, Termination

- We are going to sacrifice completeness

=> False positives but no false negatives

What False Positives Mean

x MAY-ALIAS z?

No

```
Circle x = new Circle();
```

```
Circle z = new Circle();
```

[x != z]

```
x.radius = 1;
```

[x.radius == 1, x != z]

```
z.radius = 2;
```

[x.radius == 1]

```
y = x.radius;
```

[y == 1]

```
assert(y == 1)
```

Yes

[x == z or x != z]

[x.radius == 1, x == z or x != z]

[x.radius == 1 or x.radius == 2]

[y == 1 or y == 2]

False Positive!

Pointer analysis answers questions of form: MayAlias(x, z)?

No => x and z are not aliased in any run

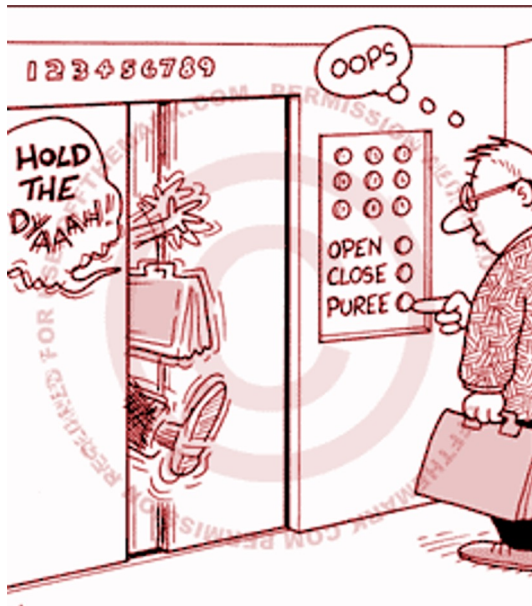
Yes => Can't tell if x and z are aliased in some run

Approximation to the Rescue

- Many sound approximate algorithms for pointer analysis
- Varying levels of precision
- Differ in two key aspects:
 - How to abstract the **heap** (i.e. dynamically allocated data)
 - How to abstract control-flow

Example Java Program

```
class Elevator {  
    Object[] floors;  
    Object[] events;  
}
```

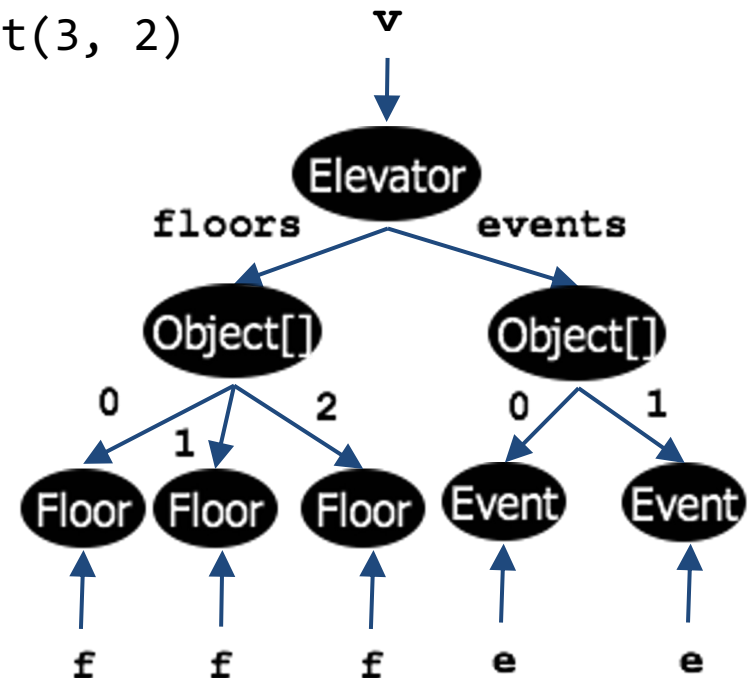


```
void doit(int M, int N) {  
    Elevator v = new Elevator();  
  
    v.floors = new Object[M];  
    v.events = new Object[N];  
  
    for (int i = 0; i < M; i++) {  
        Floor f = new Floor();  
        v.floors[i] = f;  
    }  
  
    for (int i = 0; i < N; i++) {  
        Event e = new Event();  
        v.events[i] = e;  
    }  
}
```

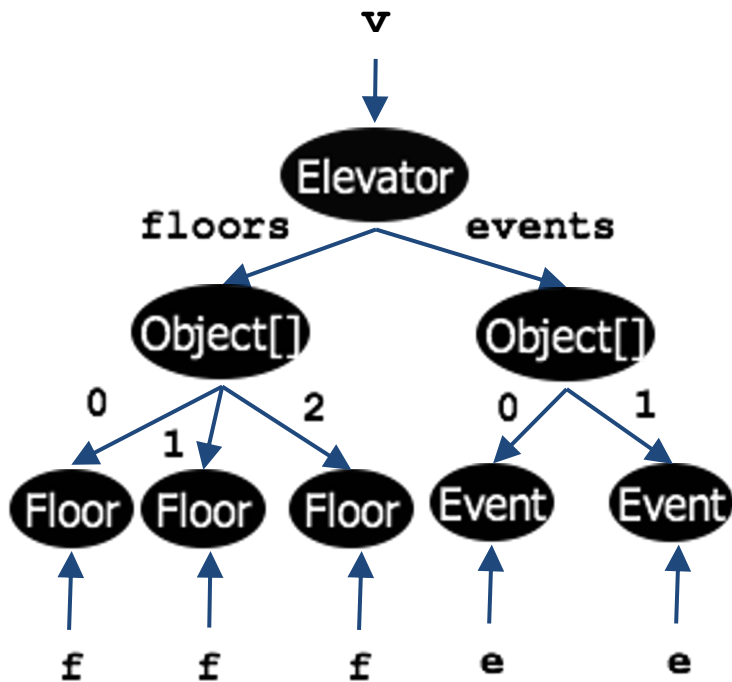
A Run of the Program

```
void doit(int M, int N) {  
    Elevator v = new Elevator();  
  
    v.floors = new Object[M];  
    v.events = new Object[N];  
  
    for (int i = 0; i < M; i++) {  
        Floor f = new Floor();  
        v.floors[i] = f;  
    }  
  
    for (int i = 0; i < N; i++) {  
        Event e = new Event();  
        v.events[i] = e;  
    }  
}
```

doit(3, 2)

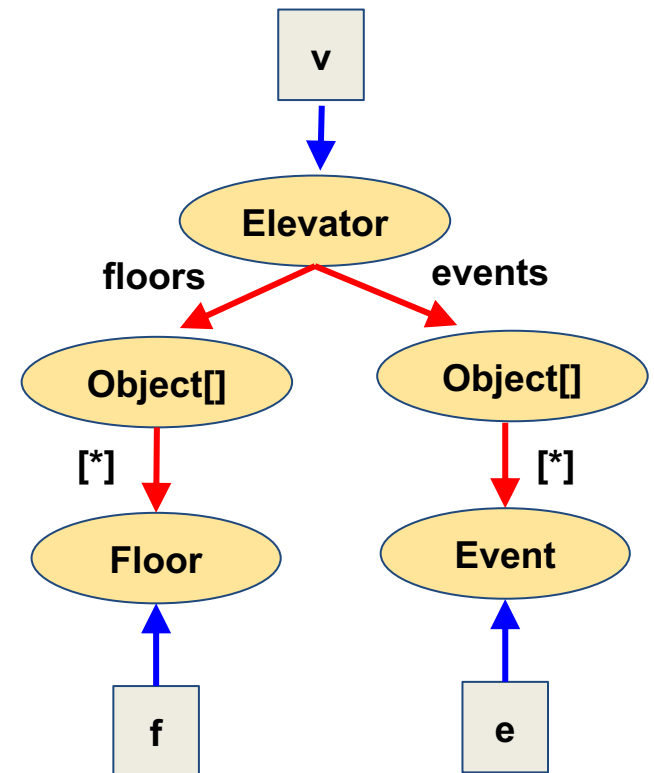
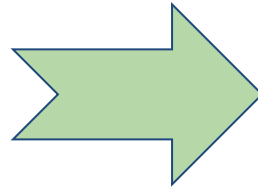
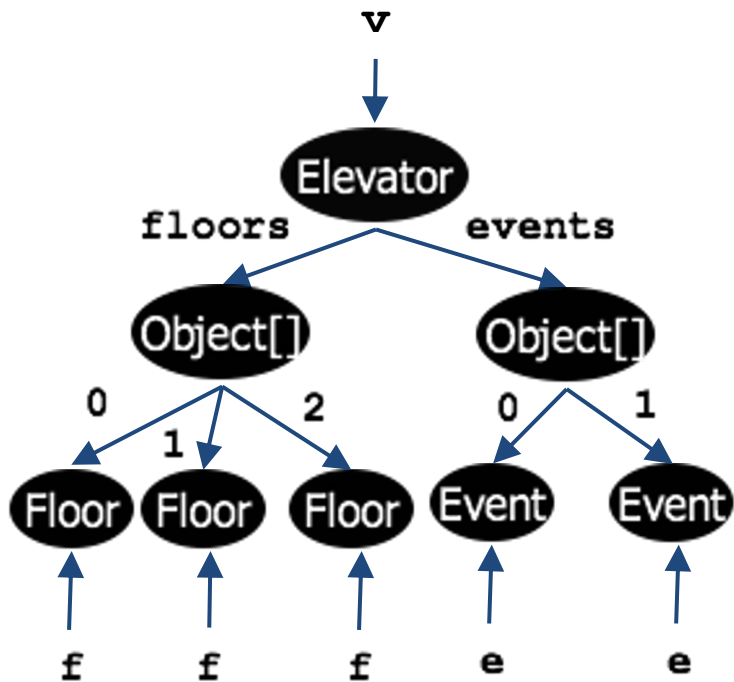


Abstracting the Heap



```
void doit(int M, int N) {  
    Elevator v = new Elevator();  
  
    v.floors = new Object[M];  
    v.events = new Object[N];  
  
    for (int i = 0; i < M; i++) {  
        Floor f = new Floor();  
        v.floors[i] = f;  
    }  
  
    for (int i = 0; i < N; i++) {  
        Event e = new Event();  
        v.events[i] = e;  
    }  
}
```

Result of Heap Abstraction: Points-to Graph



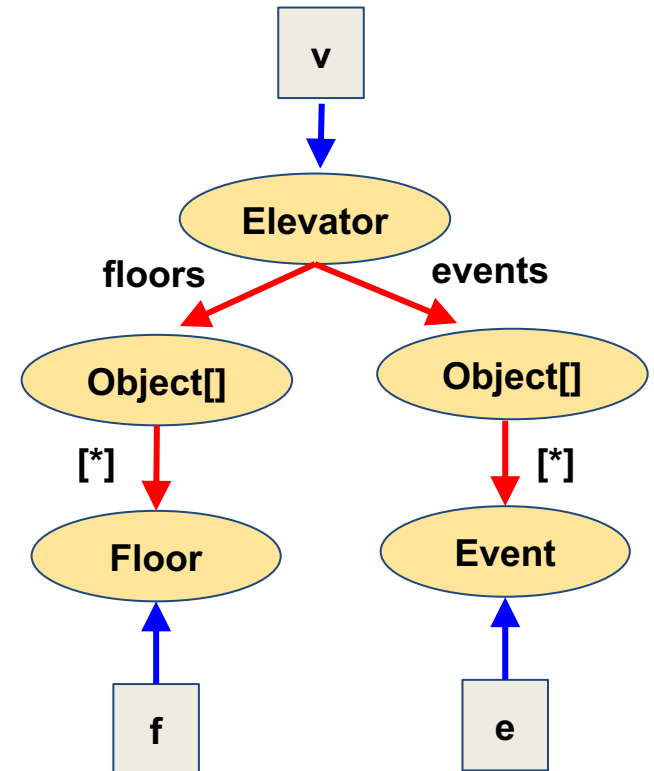
variable



allocation site

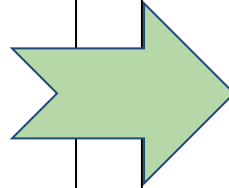
Abstracting Control-Flow

```
void doit(int M, int N) {  
    Elevator v = new Elevator();  
  
    v.floors = new Object[M];  
    v.events = new Object[N];  
  
    for (int i = 0; i < M; i++) {  
        Floor f = new Floor();  
        v.floors[i] = f;  
    }  
  
    for (int i = 0; i < N; i++) {  
        Event e = new Event();  
        v.events[i] = e;  
    }  
}
```



Flow Insensitivity

```
void doit(int M, int N) {  
    Elevator v = new Elevator();  
  
    v.floors = new Object[M];  
    v.events = new Object[N];  
  
    for (int i = 0; i < M; i++) {  
        Floor f = new Floor();  
        v.floors[i] = f;  
    }  
  
    for (int i = 0; i < N; i++) {  
        Event e = new Event();  
        v.events[i] = e;  
    }  
}
```



```
void doit(int M, int N) {  
    v = new Elevator  
  
    v.floors = new Object[]  
    v.events = new Object[]  
  
    f = new Floor  
    v.floors[*] = f  
  
    e = new Event  
    v.events[*] = e  
}
```

Chaotic Iteration Algorithm

graph = empty

repeat:

 for (each statement *s* in set)

 apply rule corresponding to *s* on graph

until graph stops changing

Kinds of Statements

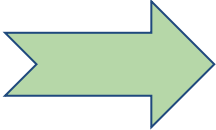
(statement) $s ::= v = \text{new } \dots \mid v = v2 \mid v2 = v.f \mid$
 $v.f = v2 \mid v2 = v[*] \mid v[*] = v2$

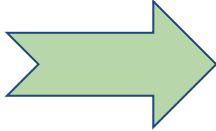
(pointer-type variable) v

(pointer-type field) f

Is This Grammar Enough?

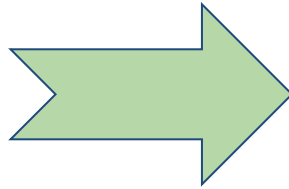
<code>v = new ...</code>		<code>v = v2</code>		<code>v2 = v.f</code>	
<code>v.f = v2</code>		<code>v2 = v[*]</code>		<code>v[*] = v2</code>	

<code>v.events = new Object[]</code>		<code>tmp = new Object[]</code>
		<code>v.events = tmp</code>

<code>v.events[*] = e</code>		<code>tmp = v.events</code>
		<code>tmp[*] = e</code>

Example Program in Normal Form

```
void doit(int M, int N) {  
    v = new Elevator  
  
    v.floors = new Object[]  
    v.events = new Object[]  
  
    f = new Floor  
    v.floors[*] = f  
  
    e = new Event  
    v.events[*] = e  
  
}
```



```
void doit(int M, int N) {  
    v = new Elevator  
  
    tmp1 = new Object[]  
    v.floors = tmp1  
    tmp2 = new Object[]  
    v.events = tmp2  
  
    f = new Floor  
    tmp3 = v.floors  
    tmp3[*] = f  
  
    e = new Event  
    tmp4 = v.events  
    tmp4[*] = e  
  
}
```

QUIZ: Normal Form of Programs

$v = \text{new } \dots$		$v = v2$		$v2 = v.f$	
$v.f = v2$		$v2 = v[*]$		$v[*] = v2$	

Convert each of these two expressions to normal form:

$v1.f = v2.f$



$v1.f.g = v2.h$

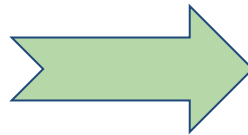


QUIZ: Normal Form of Programs

$v = \text{new } \dots$		$v = v2$		$v2 = v.f$	
$v.f = v2$		$v2 = v[*]$		$v[*] = v2$	

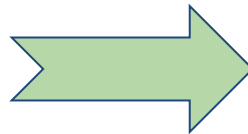
Convert each of these two expressions to normal form:

$v1.f = v2.f$



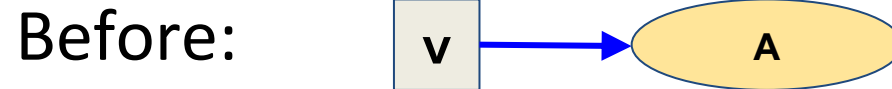
```
tmp = v2.f
v1.f = tmp
```

$v1.f.g = v2.h$

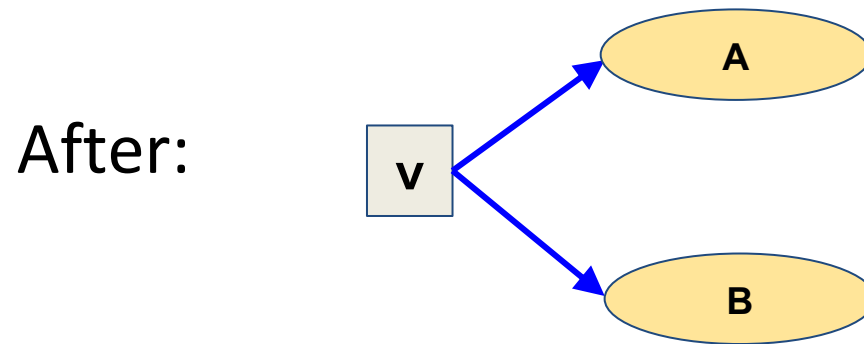


```
tmp1 = v1.f
tmp2 = v2.h
tmp1.g = tmp2
```


Rule for Object Allocation Sites

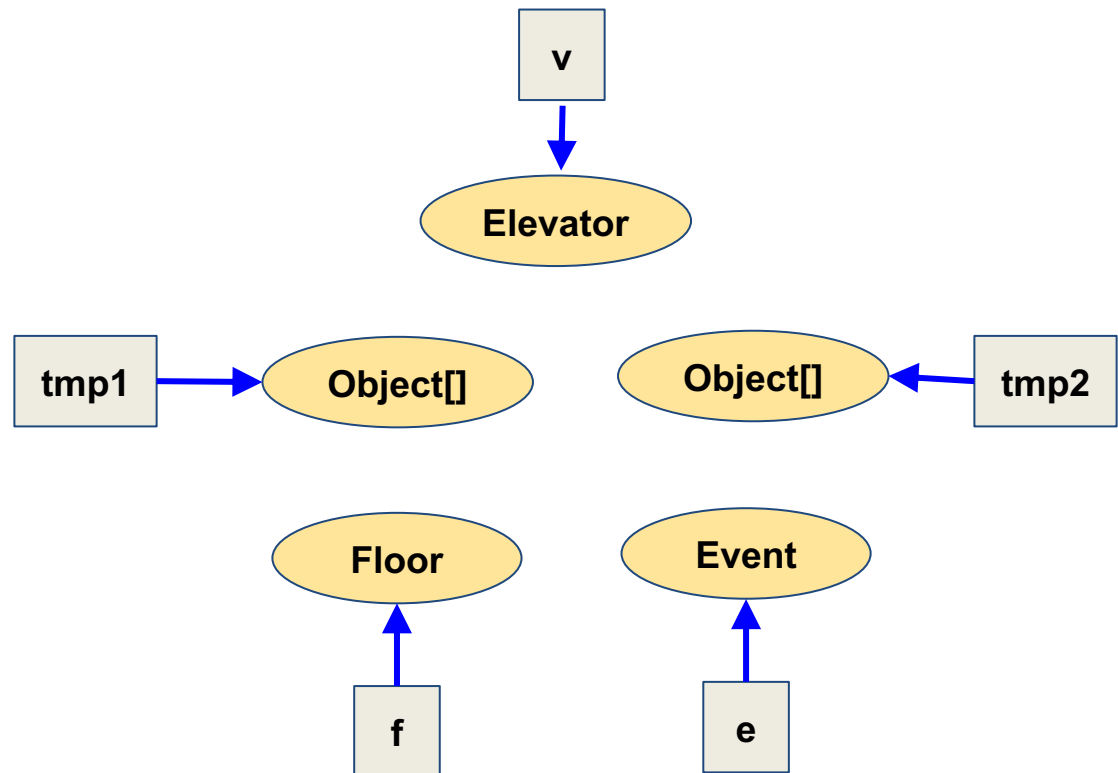


v = new B



Rule for Object Allocation Sites: Example

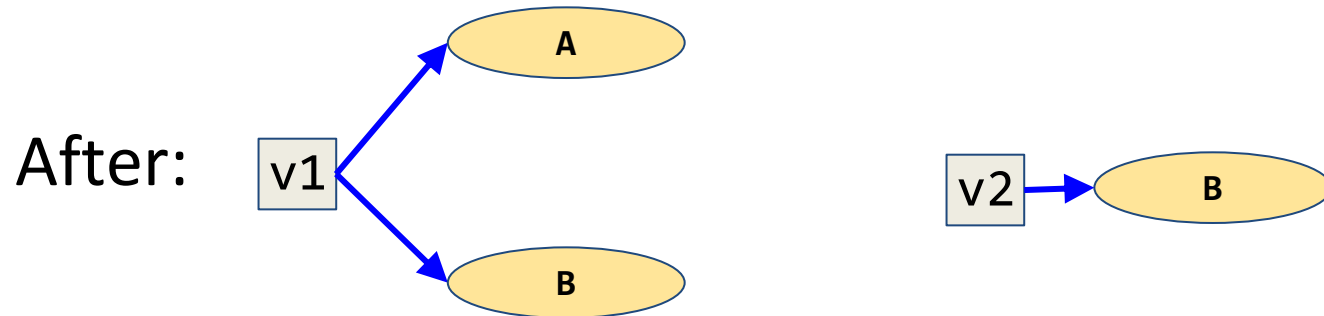
```
void doit(int M, int N) {  
    v = new Elevator  
  
    tmp1 = new Object[]  
    v.floors = tmp1  
    tmp2 = new Object[]  
    v.events = tmp2  
  
    f = new Floor  
    tmp3 = v.floors  
    tmp3[*] = f  
  
    e = new Event  
    tmp4 = v.events  
    tmp4[*] = e  
}
```



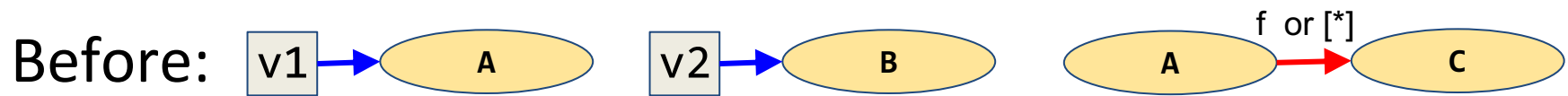
Rule for Object Copy



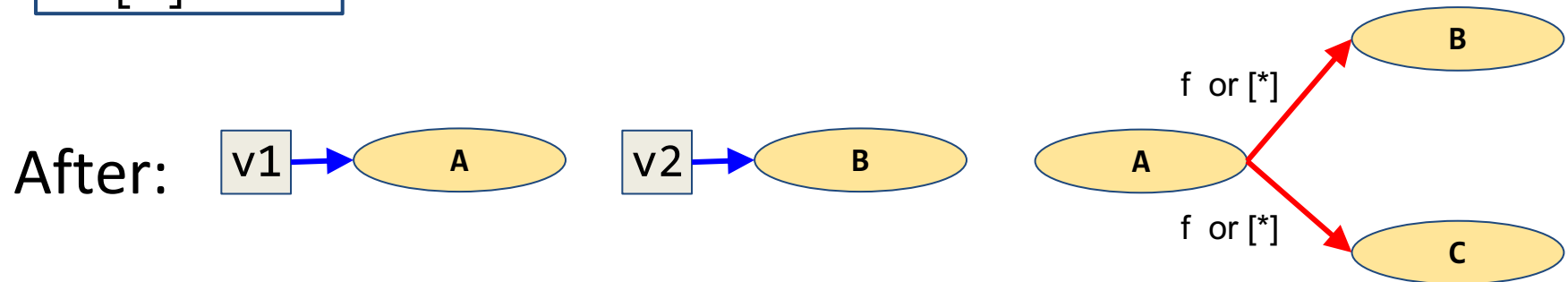
$v1 = v2$



Rule for Field Writes

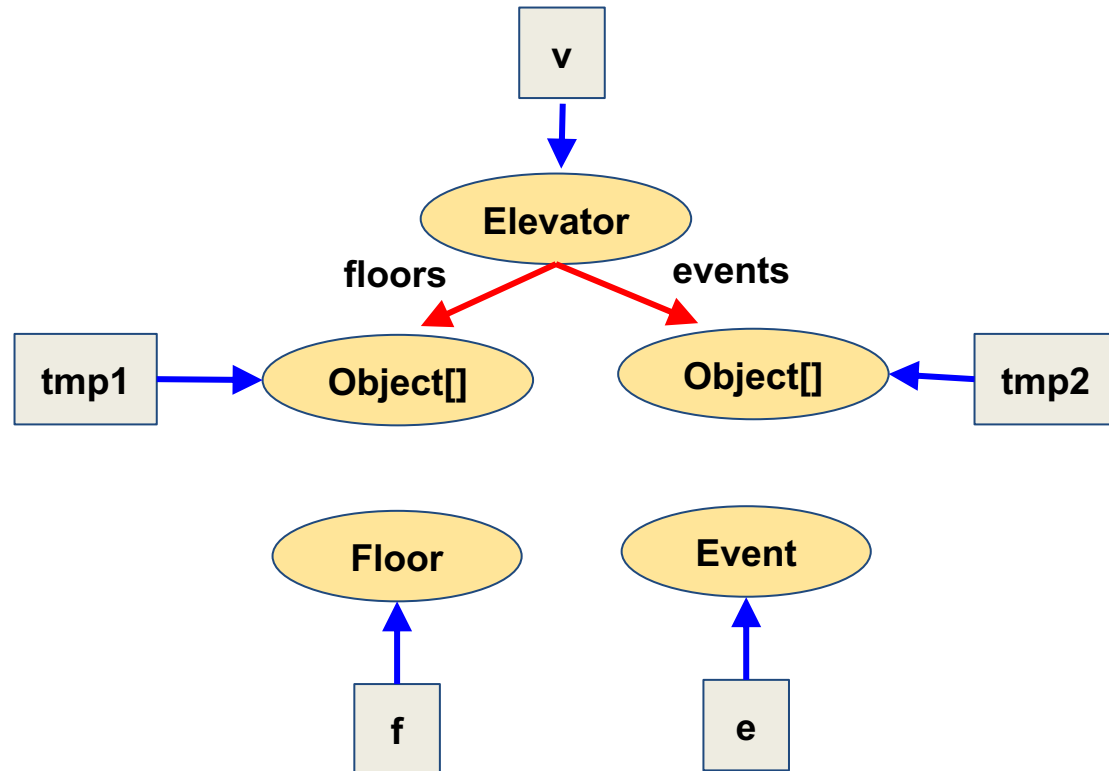


$v1 . f = v2$
or
 $v1[*] = v2$

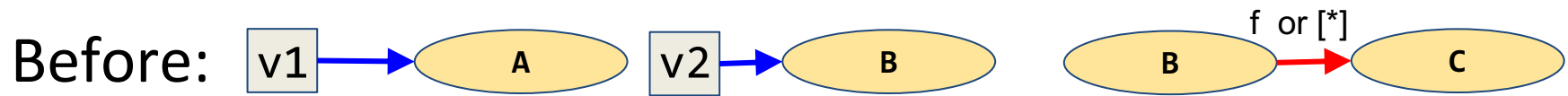


Rule for Field Writes: Example

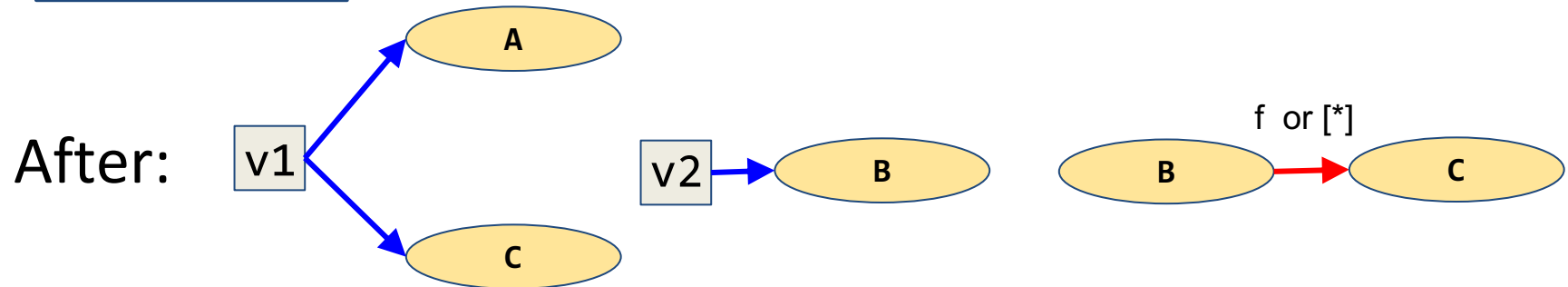
```
void doit(int M, int N) {  
    v = new Elevator  
  
    tmp1 = new Object[]  
    v.floors = tmp1  
    tmp2 = new Object[]  
    v.events = tmp2  
  
    f = new Floor  
    tmp3 = v.floors  
    tmp3[*] = f  
  
    e = new Event  
    tmp4 = v.events  
    tmp4[*] = e  
}
```



Rule for Field Reads

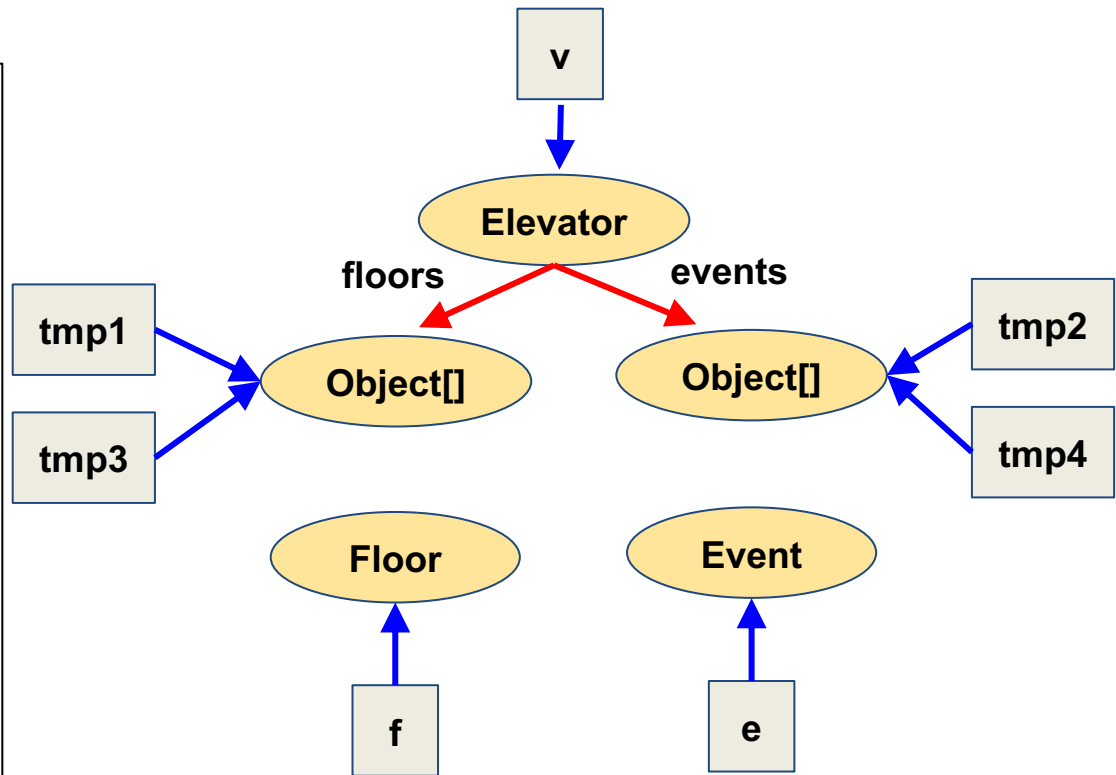


$v1 = v2.f$
or
 $v1 = v2[*]$



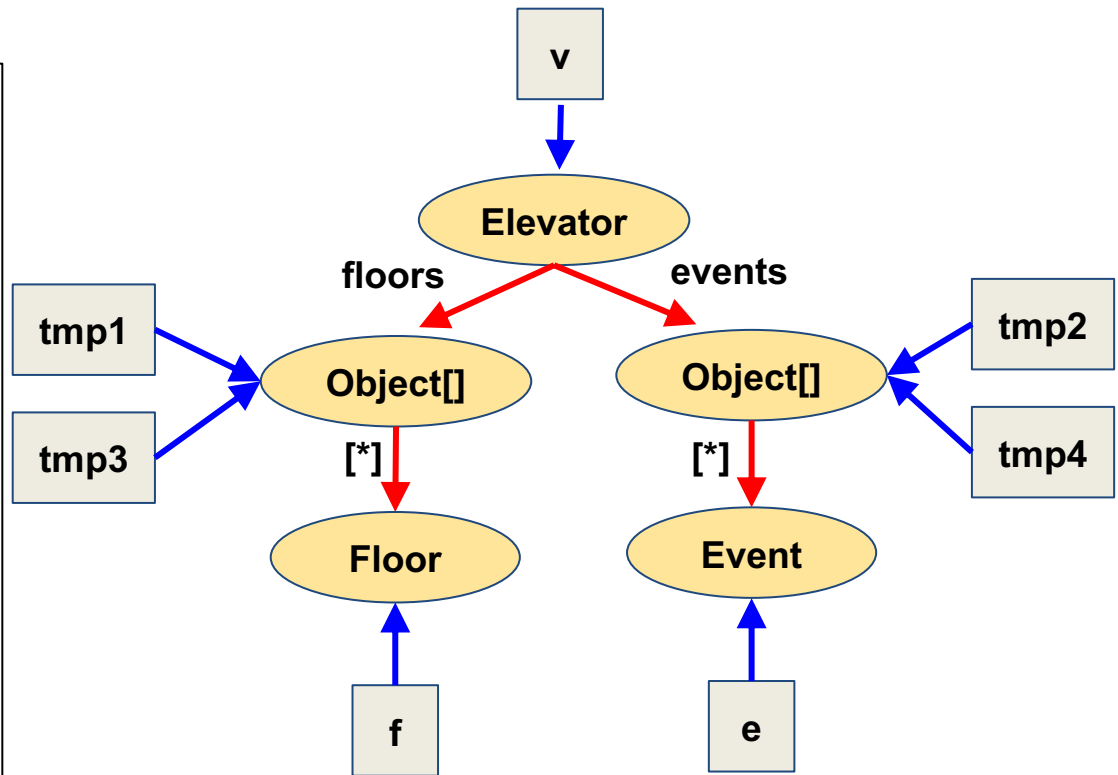
Rule for Field Reads: Example

```
void doit(int M, int N) {  
    v = new Elevator  
  
    tmp1 = new Object[]  
    v.floors = tmp1  
    tmp2 = new Object[]  
    v.events = tmp2  
  
    f = new Floor  
    tmp3 = v.floors  
    tmp3[*] = f  
  
    e = new Event  
    tmp4 = v.events  
    tmp4[*] = e  
}
```



Continuing the Pointer Analysis: Example

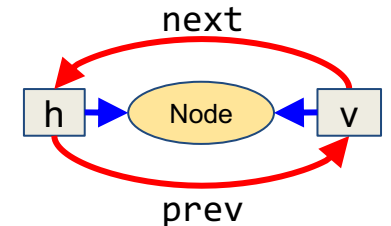
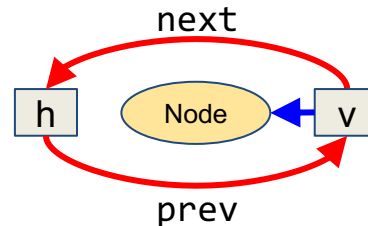
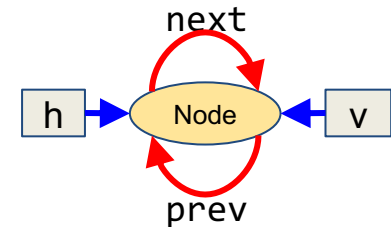
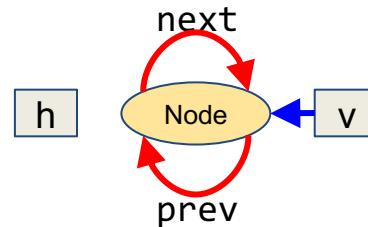
```
void doit(int M, int N) {  
    v = new Elevator  
  
    tmp1 = new Object[]  
    v.floors = tmp1  
    tmp2 = new Object[]  
    v.events = tmp2  
  
    f = new Floor  
    tmp3 = v.floors  
    tmp3[*] = f  
  
    e = new Event  
    tmp4 = v.events  
    tmp4[*] = e  
}
```



QUIZ: Pointer Analysis Example

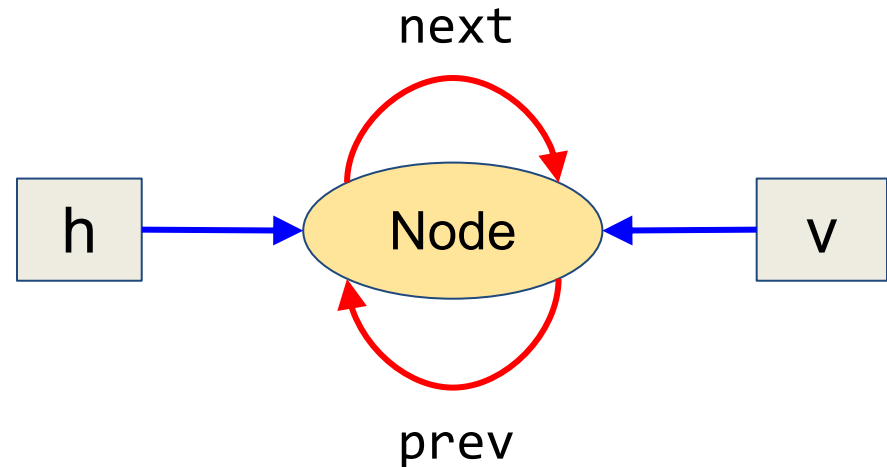
```
class Node {  
    int data;  
    Node next, prev;  
}  
  
Node h = null;  
for (...) {  
    Node v = new Node();  
    if (h != null) {  
        v.next = h;  
        h.prev = v;  
    }  
    h = v;  
}
```

Choose the points-to graph for the shown program.



QUIZ: Pointer Analysis Example

```
class Node {  
    int data;  
    Node next, prev;  
}  
  
Node h = null;  
for (...) {  
    Node v = new Node();  
    if (h != null) {  
        v.next = h;  
        h.prev = v;  
    }  
    h = v;  
}
```



Classifying Pointer Analysis Algorithms

- Is it flow-sensitive?
- Is it context-sensitive?
- What heap abstraction scheme is used?
- How are aggregate data types modeled?

Flow Sensitivity

- How to model control-flow **within** a procedure
- Two kinds: flow-insensitive vs. flow-sensitive
- Flow-insensitive == **weak updates**
 - Suffices for may-alias analysis
- Flow-sensitive == **strong updates**
 - Required for must-alias analysis

Context Sensitivity

- How to model control-flow **across** procedures
- Two kinds: context-insensitive vs. context-sensitive
- Context-insensitive: analyze each procedure once
- Context-sensitive: analyze each procedure possibly multiple times, once per abstract calling context

Heap Abstraction

- Scheme to partition unbounded set of concrete objects into finitely many **abstract objects** (oval nodes in points-to graph)
- Ensures that pointer analysis terminates
- Many sound schemes, varying in precision & efficiency
 - Too few abstract objects => efficient but imprecise
 - Too many abstract objects => expensive but precise

Scheme #1: Allocation-Site Based

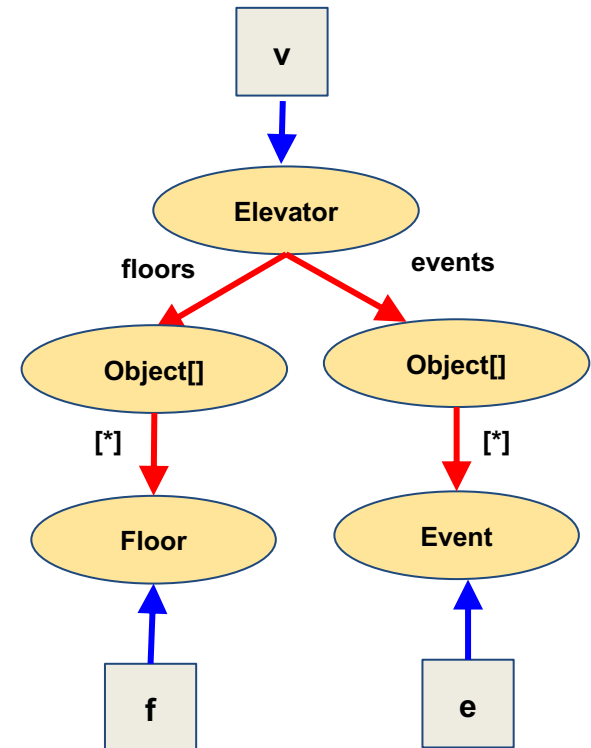
One abstract object per **allocation site**

Allocation site identified by:

- **new** keyword in Java/C++
- **malloc()** call in C

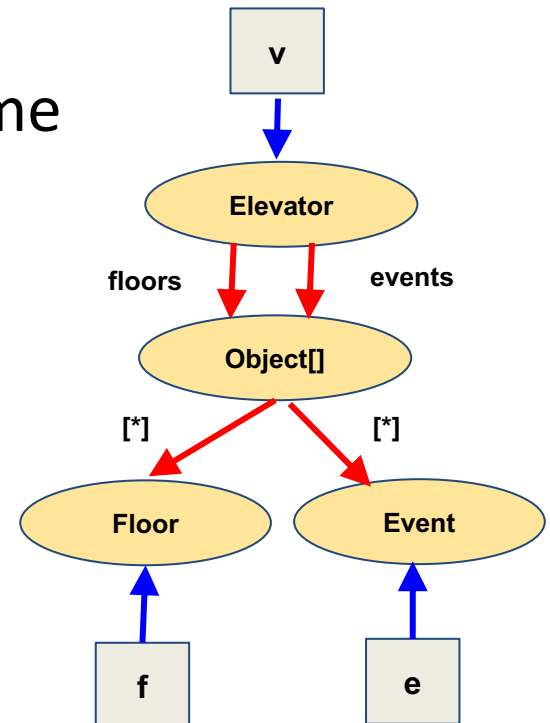
Finitely many allocation sites in a program

=> finitely many abstract objects



Scheme #2: Type Based

- Allocation-site based scheme can be costly
 - Large programs
 - Clients needing quick turnaround time
 - Overly fine granularity of sites
- One abstract object per **type**
- Finitely many types in a program
=> finitely many abstract objects

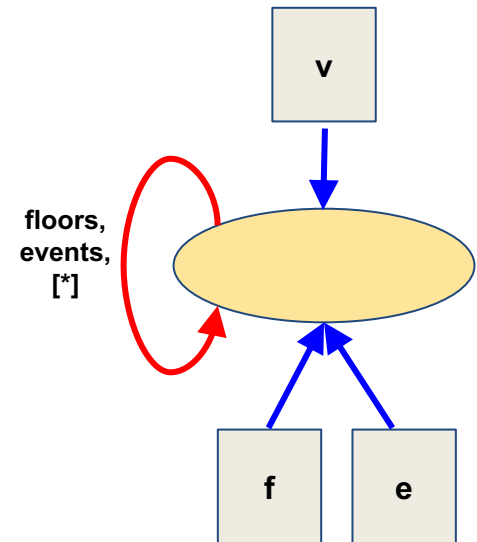


Scheme #3: Heap-Insensitive

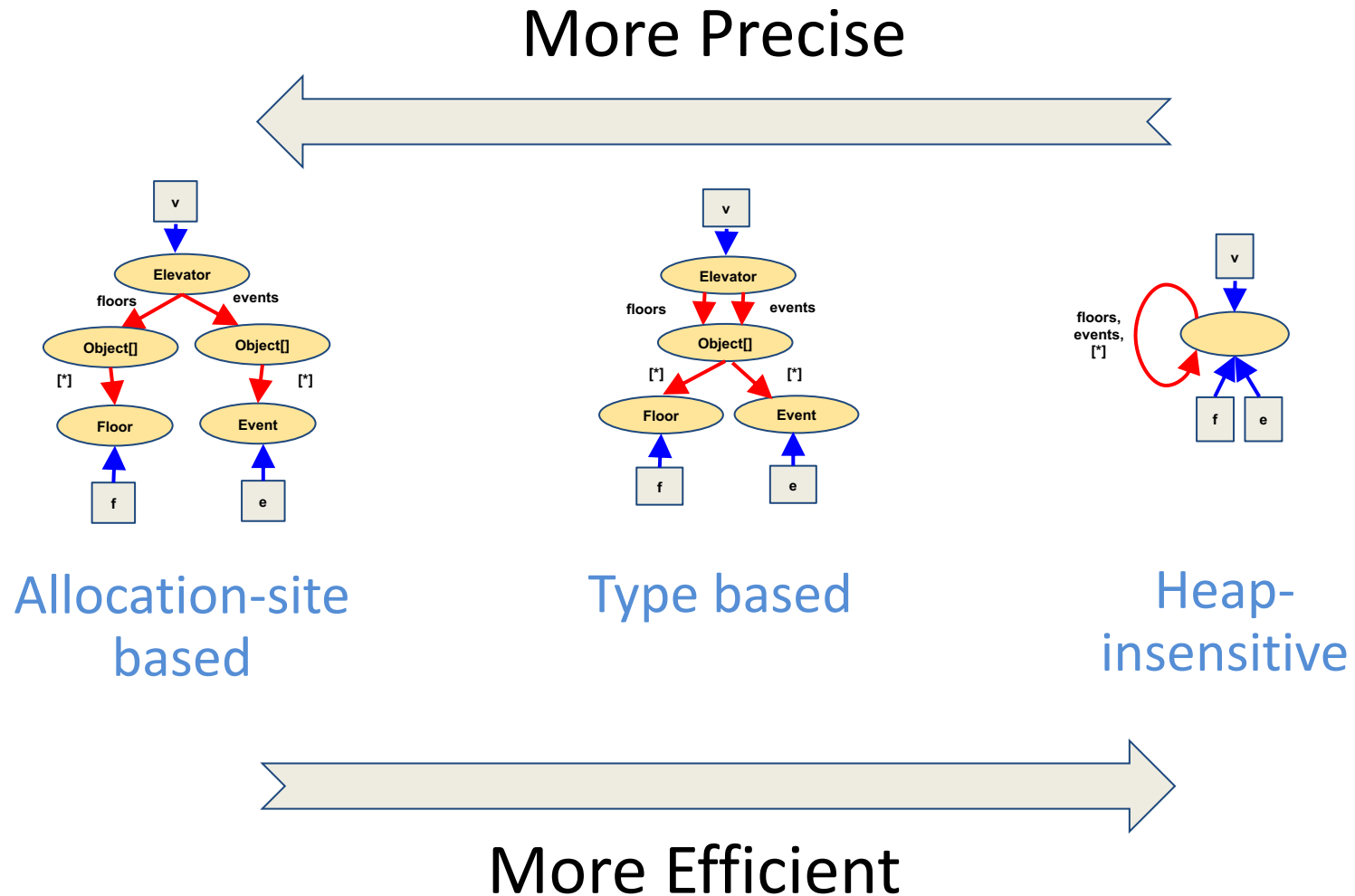
Single abstract object representing entire heap

Popular for languages with primarily stack-directed pointers (e.g. C)

Unsuitable for languages with only heap-directed pointers (e.g. Java)



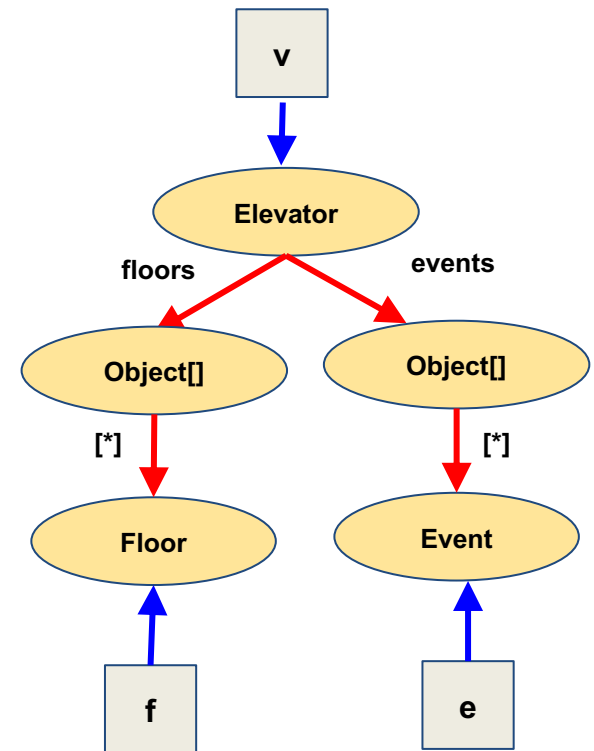
Tradeoffs in Heap Abstraction Schemes



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

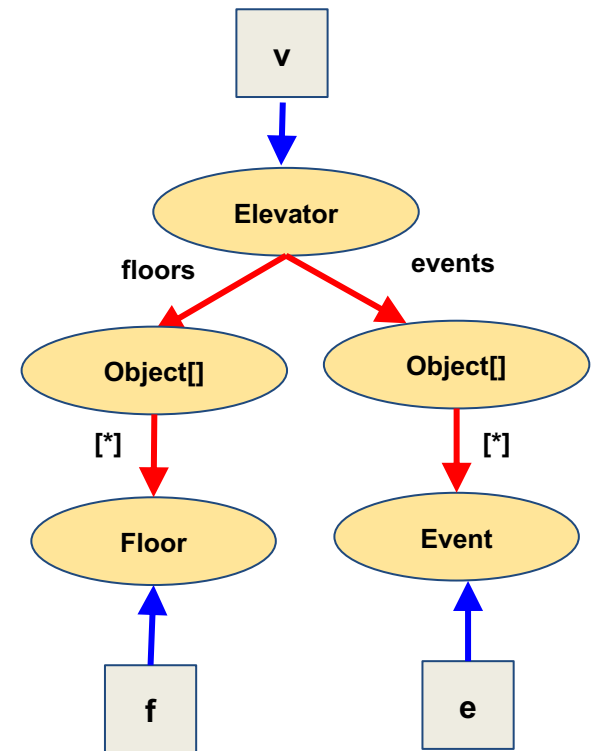
May-Alias?	Allocation-Site Based	Type Based
e, f	No	
v.floors, v.events		
v.floors[0], v.events[0]		
v.events[0], v.events[2]	Yes	



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

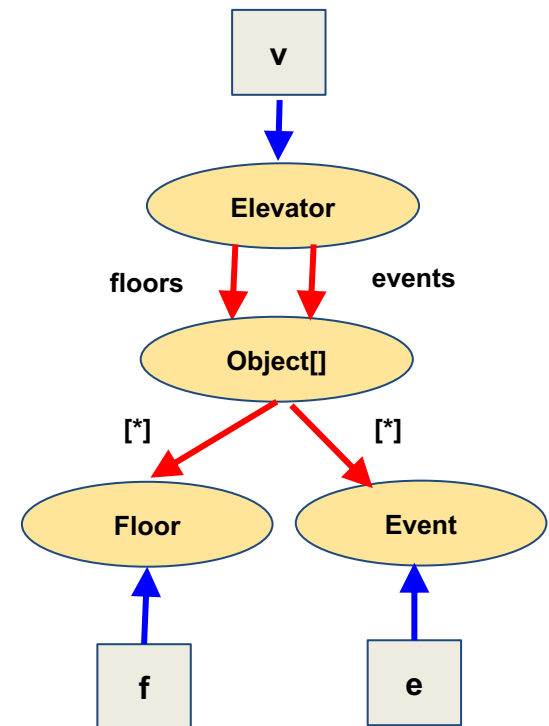
May-Alias?	Allocation-Site Based	Type Based
e, f	No	
v.floors, v.events	No	
v.floors[0], v.events[0]	No	
v.events[0], v.events[2]	Yes	



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

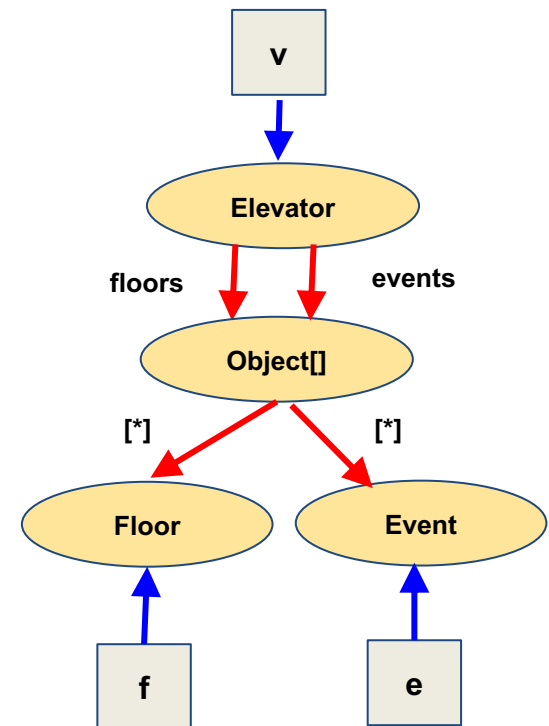
May-Alias?	Allocation-Site Based	Type Based
e, f	No	
v.floors, v.events	No	
v.floors[0], v.events[0]	No	
v.events[0], v.events[2]	Yes	



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

May-Alias?	Allocation-Site Based	Type Based
e, f	No	No
v.floors, v.events	No	Yes
v.floors[0], v.events[0]	No	Yes
v.events[0], v.events[2]	Yes	Yes



Modeling Aggregate Data Types: Arrays

- Common choice: single field `[*]` to represent all array elements
 - Cannot distinguish different elements of same array
- More sophisticated representations that make such distinctions are employed by array dependence analyses
 - Used to parallelize sequential loops by parallelizing compilers

Modeling Aggregate Data Types: Records

Three choices:

1. **Field-insensitive**: merge **all** fields of **each** record object
2. **Field-based**: merge **each** field of **all** record objects
3. **Field-sensitive**: keep **each** field of **each** (abstract) record object separate

	f1	f2
a1	Red	Red
a2	Blue	Blue

	f1	f2
a1	Red	Blue
a2	Red	Blue

	f1	f2
a1	Red	Yellow
a2	Pink	Blue

QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned in this lesson.

Flow-sensitive?	
Context-sensitive?	
Distinguishes fields of object?	
Distinguishes elements of array?	
What kind of heap abstraction?	

A. Yes

B. No

A. Yes

B. No

A. Yes

B. No

A. Yes

B. No

A. Allocation-
site based

B. Type
based

QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned in this lesson.

Flow-sensitive?	B
Context-sensitive?	B
Distinguishes fields of object?	A
Distinguishes elements of array?	B
What kind of heap abstraction?	A

A. Yes

B. No

A. Yes

B. No

A. Yes

B. No

A. Yes

B. No

A. Allocation-
site based

B. Type
based

What Have We Learned?

- What is pointer analysis?
- May-alias analysis vs. must-alias analysis
- Points-to graphs
- Working of a pointer analysis algorithm
- Classifying pointer analyses: flow sensitivity, context sensitivity, heap abstraction, aggregate modeling