Pointer Analysis

Introduction

1. Learn about how to reason about the flow of non-primitive data
   - Better known as pointers, objects, or references
   - Called pointer analysis
   - Pointers are prevalent in C, C++, Java, Python
   - Pointer analysis is fundamental to any static analysis for reasoning about the flow of data in programs written today

Introducing Pointers

1. Example without pointers

```
x = 1;
y = x;
assert(y == 1)
```

2. Same example with pointers
   - Requires tracking expressions, not just variables

```
x = new Circle();
x.radius = 1; // field write
y = x.radius; // field read
assert(y == 1);
```

Pointer Aliasing

1. Pointer aliasing: Situation in which same address referred to in different ways

```
Circle x = new Circle();
Circle z = ?;
x.radius = 1;
z.radius = 2; // don't know if z is an alias of x
y = x.radius;
assert(y == 1)
```

May-Alias Analysis

1. Suppose x and z denote different circles
   - We can infer that the value of x.radius remains 1 even after z is assigned 2
   - Infer that y == 1 because x.radius is 1
     - Can prove this assertion by tracking the fact that it is not true that x and z may alias
   - May-Alias Analysis == Pointer Analysis

```
Circle x = new Circle();
Circle z = new Circle();
x.radius = 1;
z.radius = 2; // don't know if z is an alias of x
y = x.radius;
assert(y == 1)
```

Must-Alias Analysis

1. Suppose x and z denote the same circle
   - We can infer that the value of x.radius becomes 2 after z is assigned

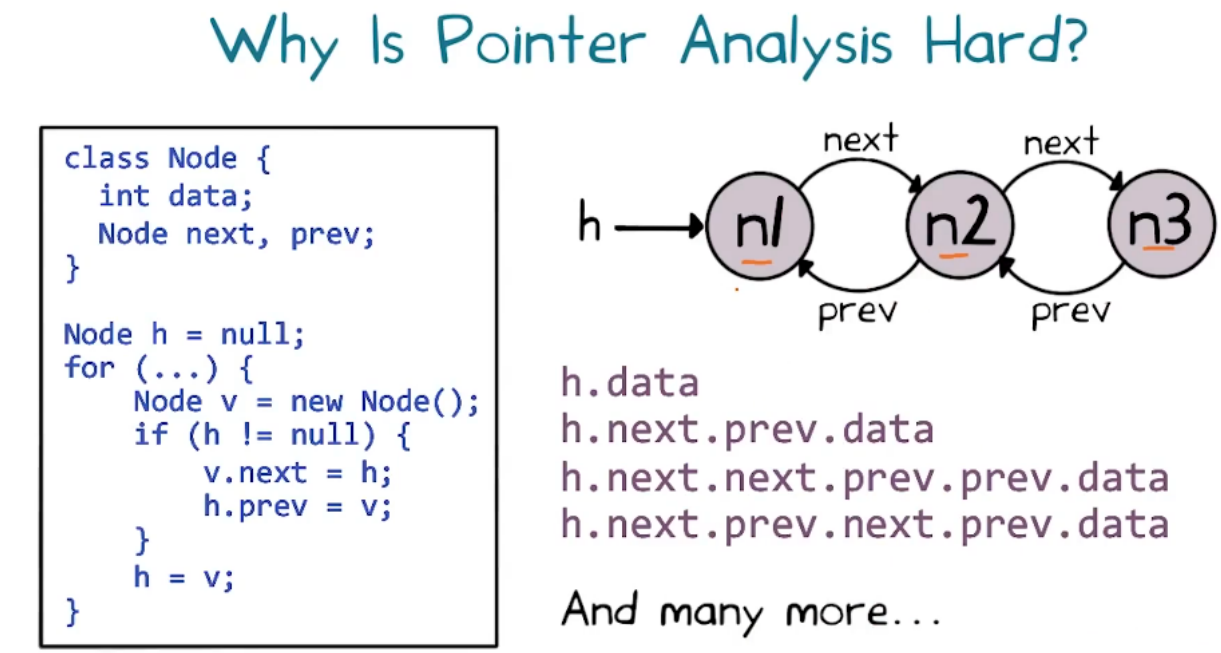
- Infer that y != 1 because x.radius is now 2
- May-Alias and Must-Alias are dual problems
  - Must-Alias is more advanced and less useful in practice

```
Circle x = new Circle();
Circle z = x;
x.radius = 1;
z.radius = 2; // don't know if z is an alias of x
y = x.radius;
assert(y == 1) // y == 2
```

Why is Pointer Analysis Hard?

1. Motivation
   - Dataflow analysis in the presence of pointers is more challenging than in the absence for a couple reasons
   - Infinitely many ways to refer to the same piece of data when cycles are present

---



Why Is Pointer Analysis Hard?

Approximation to the Rescue

1. 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
2. What False Positives Mean
   - In the above example, ask the question "x may-alias z?"
     - If no, x and z are definitely not aliases
     - If yes, x and z might be aliases, but also might not

Approximation to the Rescue 2

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

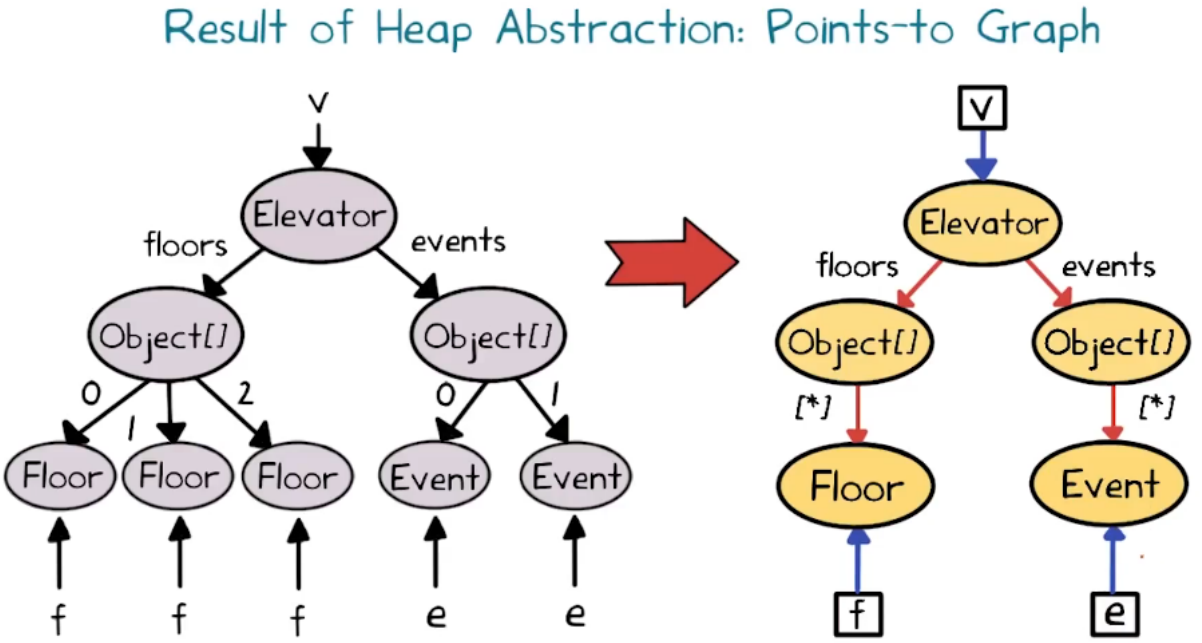
Example Java Program

1. Use this Java program throughout this lesson:

```
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;
    }
}
```

Abstracting the Heap

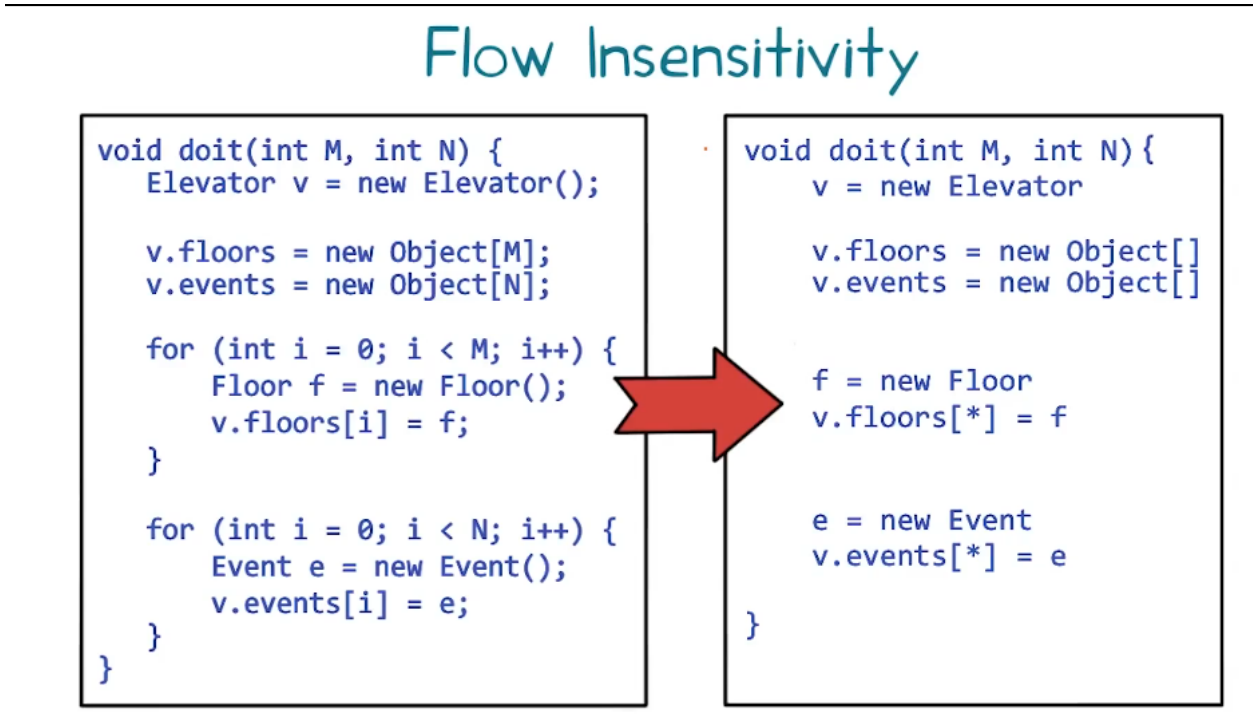
1. Pointer analysis must be able to reason about each run of this program with M floors and N events for any value of M and N
   - Pointer analysis achieves this by abstracting the heap
     - Different approaches trade off precision for efficiency
   - Different schemes
     - Abstract objects based on the site at which they're allocated in the program
   - Conflate all objects allocated in the same site to a single node in the graph
     - Single Floor node and single Event node
     - Called a points-to graph
     - Nodes:
       * Variables represented by boxes
       * Allocation sites represented by ovals
     - Edges:
       * Arrows from a variable node to an allocation site are blue
       * Arrows from an allocation site to another allocation site are red



Heap Abstraction

Abstracting Control Flow

1. Points-to graph are finite, but too expensive in practice to track a separate graph at each program point
   - Most pointer analyses only track a single points-to graph for the entire program
     - Do this by abstracting control flow
   - Flow Insensitivity
     - All control flow statements are removed
     - All statements that don't affect pointers are removed
     - Turning the program into an unordered set of statements
       - Can still prove interesting properties



Control Flow

Chaotic Iteration Algorithm

1. Algorithm
   - Oversimplification; actual algorithms use more efficient traversal techniques and data structures

```
graph = empty
Repeat:
    for (each statement s in set)
        apply rule corresponding to s on graph
    until graph stops changing
```

Kinds of Statements

1. Statements
   - Statement
     - s ::= v = new (create)
     - v = v2 (copy)
     - v2 = v.f (field read)
     - v.f = v2 (field write)
     - v = v[*] (field read, indexing into an array)
     - v[*] = v2 (field write, indexing into an array)
   - Pointer-type variable
     - v
   - Pointer-type field
     - f

Is This Grammar Enough?

1. Is this grammar sufficient for representing all operations we need to reason about for determining if pointers alias?
   - Yes; Won't prove it though
   - v.events = new Object[]
     - tmp = new Object[]
     - v.events = tmp
   - v.events[*] = e
     - tmp = v.events
     - tmp[*] = e

Example Program in Normal Form

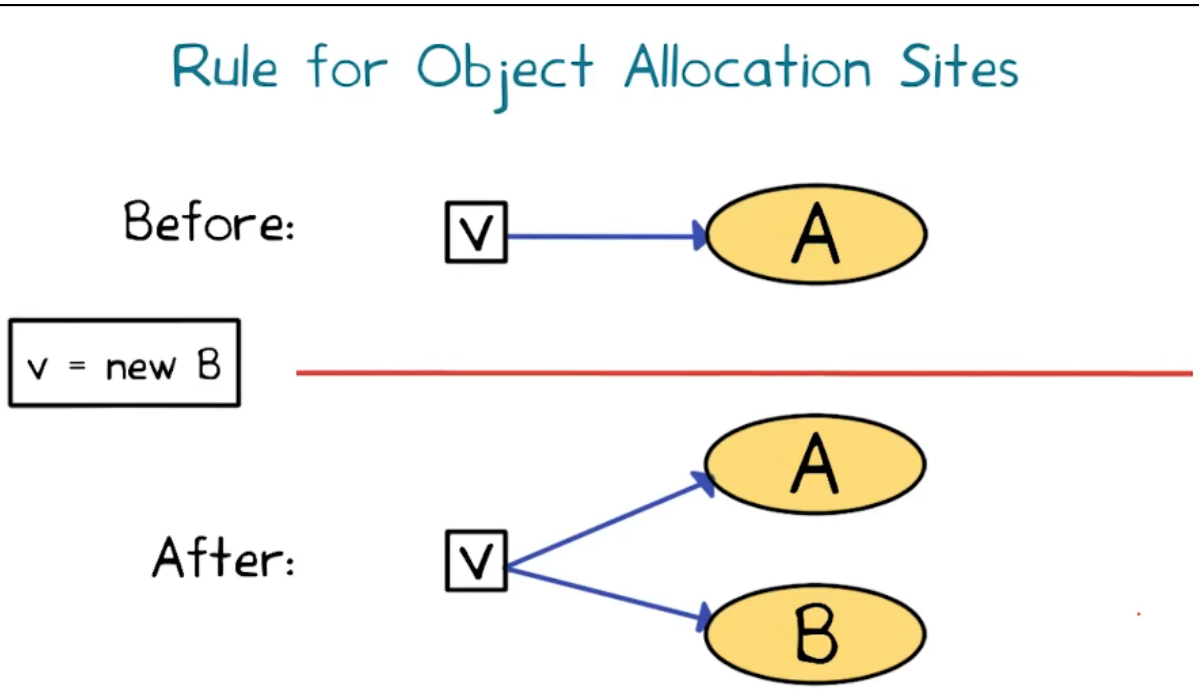
```
void doit(int M, int N) {
    v = new Elevator

    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
}
```

Normal Form of Programs

1. 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

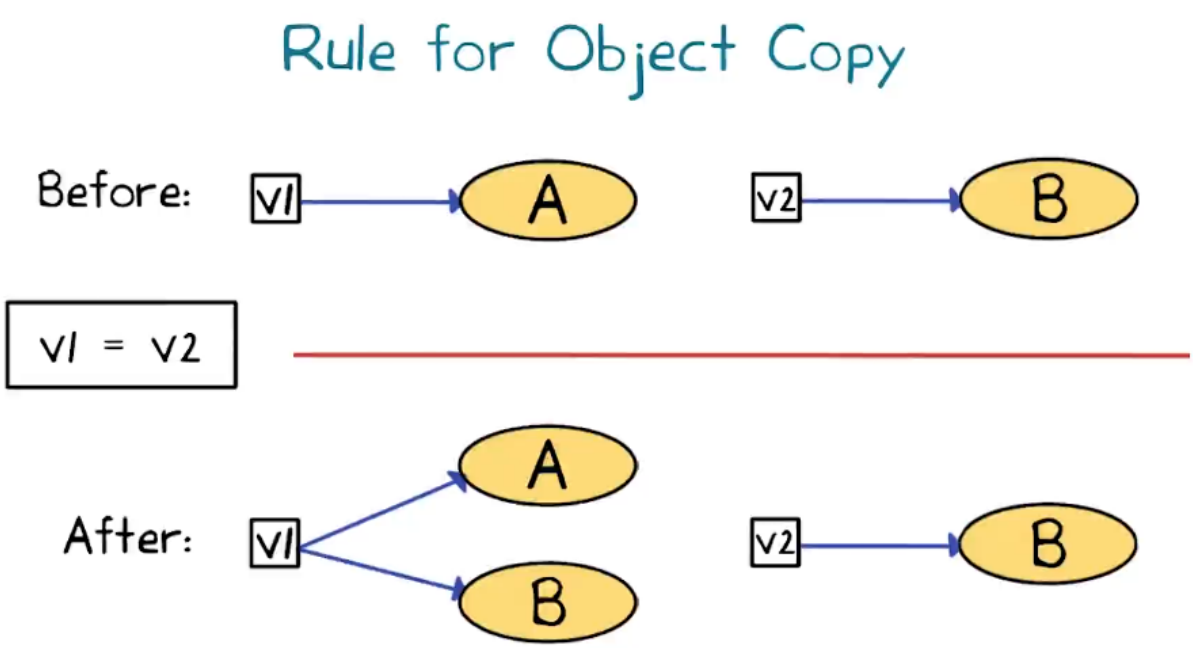
1. Need to create rules to manipulate our points-to graph for each of our six simple statements
2. v = new B
   - Create new allocation site node called B
   - Create a variable node for v if it doesn't exist
     - Add a blue arrow from variable node to allocation site
     - Weak update: Accumulate instead of replacing points-to information



Rule for Object Allocation

Rule for Object Copy

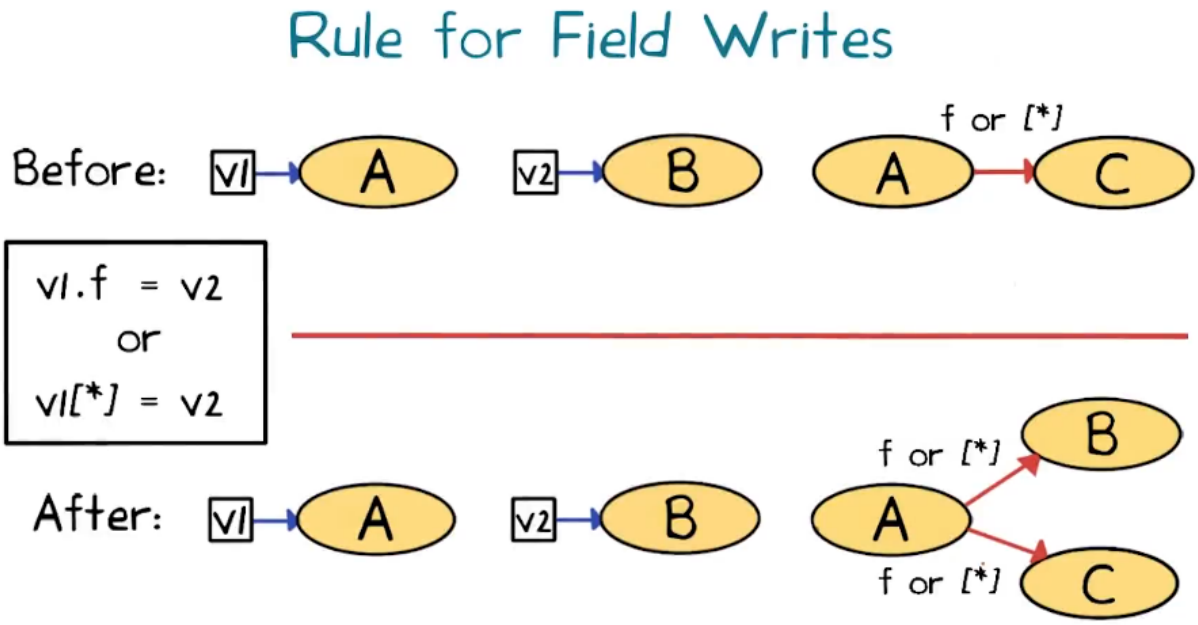
1. v1 = v2
   - Create a variable node for v1 if it doesn't exist
   - Add a blue arrow from v1 to all nodes pointed to by v2



Rule for Object Copy

Rule for Field Writes

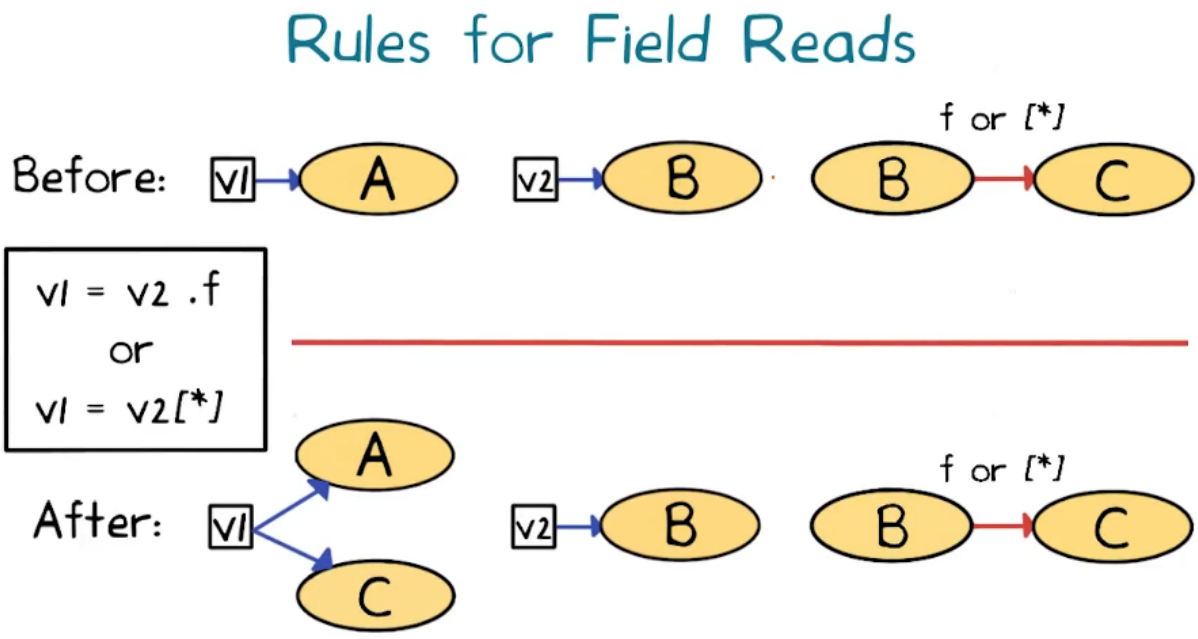
1. v1.f = v2 or v1[*] = v2
   - Add a red arrow from the variable to the allocation site



Rule for Field Writes

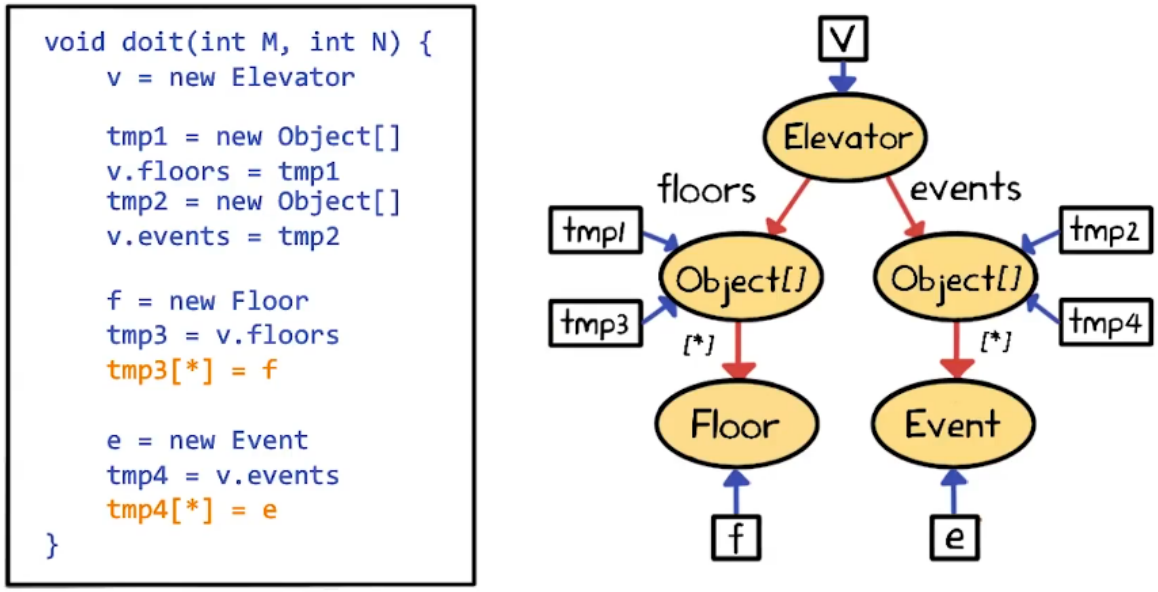
Rule for Field Reads

1. v1 = v2.f or v1 = v2[*]
   - Create a variable node for v1 if it doesn't already exist
   - Add a blue arrow from the variable to all allocation sites



Rule for Field Reads

Pointer Analysis Example

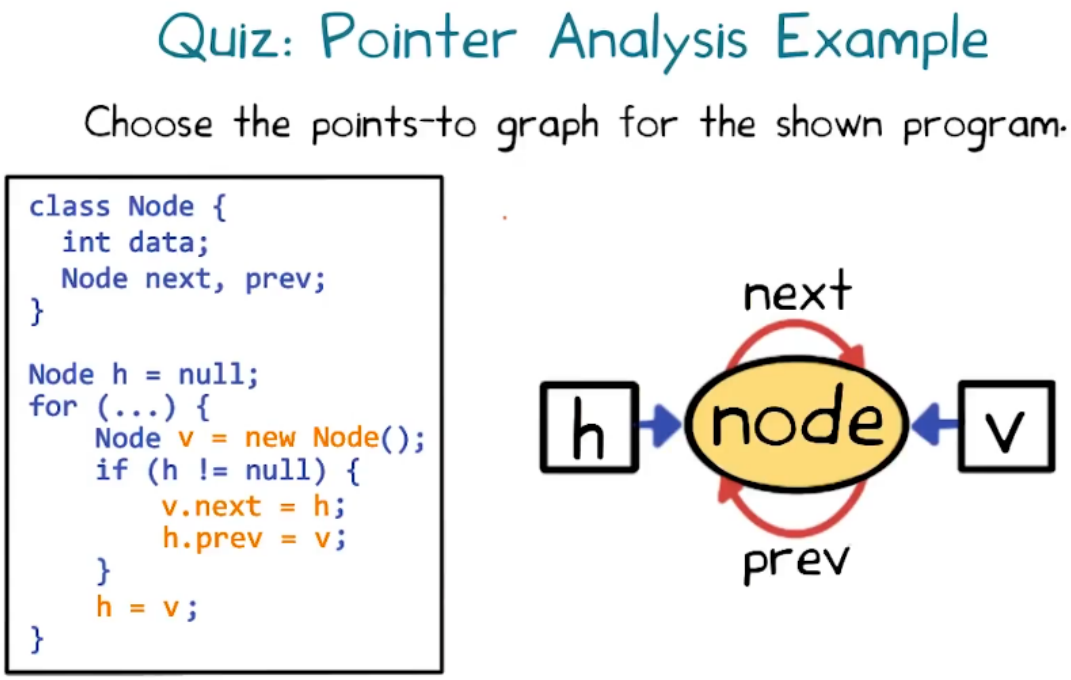


Example Graph

Pointer Analysis Example 2

1. Transform this code using the flow insensitive approximation scheme and perform chaotic iteration algorithm

---



Points-to Quiz

Classifying Pointer Analysis Algorithms

1. Four dimensions for 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

1. Intra-procedural control-flow: How to model control-flow within a procedure
   - Two kinds
     - Flow-insensitive
     - Flow-sensitive
   - Flow-insensitive == weak updates
     - Program is an unordered set of statements
     - Only generate new facts as they progress; never kill a previously generated fact
     - Suffices for may-alias analysis
   - Flow-sensitive == strong updates
     - Required for must-alias analysis
     - Capable of killing and generating facts

Context Sensitivity

1. Inter-procedural control-flow: How to model control-flow across procedures

- Two kinds
  - Context-insensitive
  - Context-sensitive
- Context-insensitive: Analyze each procedure once
  - Relatively imprecise, but very efficient
- Context-sensitive: Analyze each procedure possibly multiple times, once per abstract calling context
  - More precise but less efficient

Heap Abstraction

1. 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 and efficiency
     - Too few abstract objects -> efficient but imprecise
     - Too many abstract objects -> expensive but precise

Scheme #1: Allocation Site-Based

1. One abstract object per allocation site
   - Allocation site identified by:
     - new keyword in Java/C++
     - malloc() call in C
   - Finitely many allocation sites in programs
     - Therefore, finitely many abstract objects

Scheme #2: Type-Based

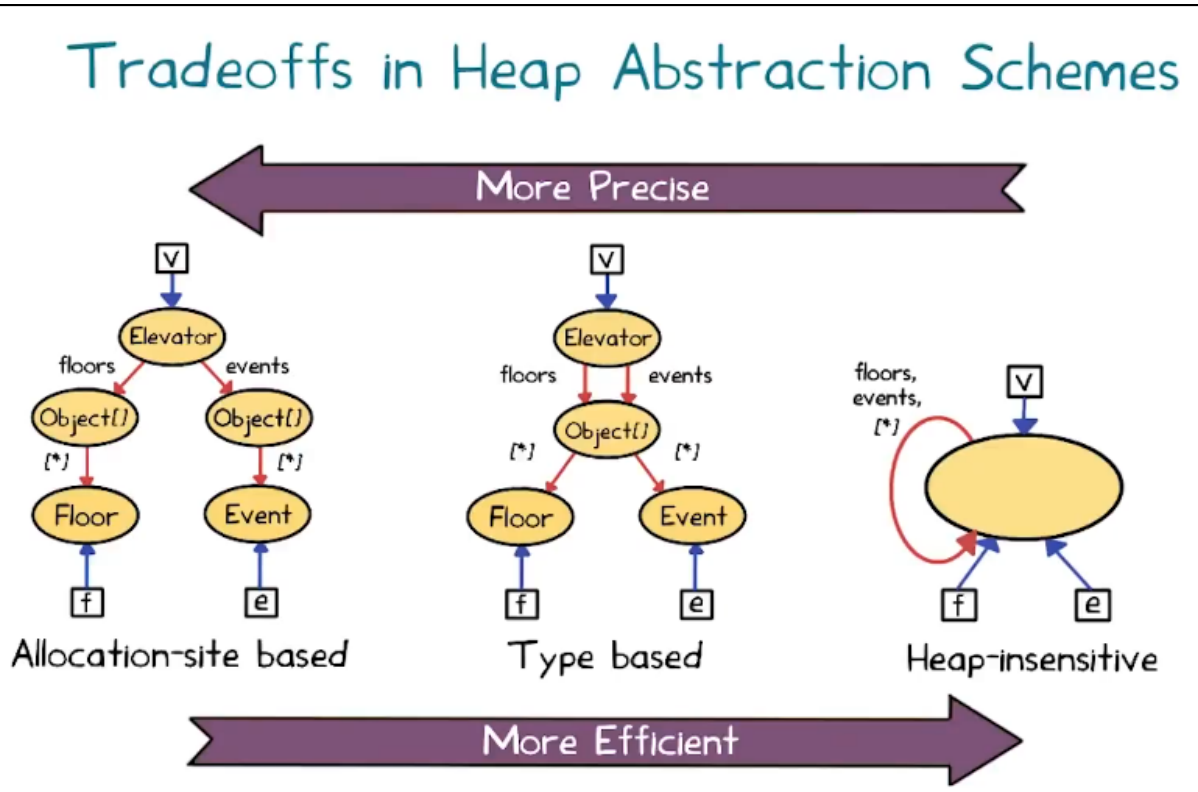
1. Allocation-site based scheme can be costly
   - Large programs
   - Clients needing quick turnaround time
   - Overly fine granularity of sites
2. Type-based Scheme
   - One abstract object per type instead of based on site
   - Finitely many types in programs
     - Therefore, finitely many abstract objects

Scheme #3: Heap-Insensitive

1. Single abstract object representing the entire heap
   - Imprecise for reasoning about the heap, but sound nevertheless
   - 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

1. There are many other schemes in the literature
   - Can design own based on analysis needs
   - Many schemes that are more expensive and precise
     - Distinguish between objects created at the same site
   - Can never have a scheme that will allow pointer analysis to make distinctions between all concrete objects and still terminate in finite time



Heap Abstraction Schemes

May-Alias Analysis

1. 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

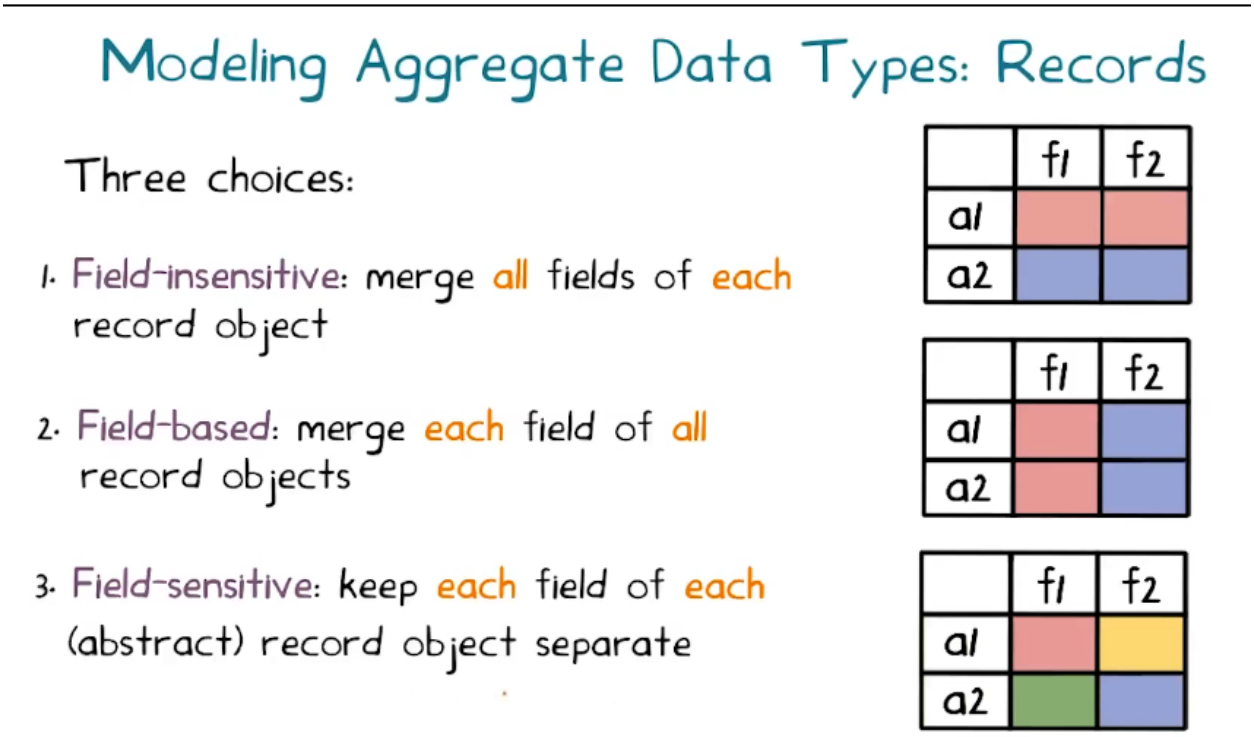
Modeling Aggregate Data Types: Arrays

1. Common choice: Single field [*] to represent all array elements
   - Cannot distinguish different elements of same array
2. 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

1. Records
   - Structs in C, classes in Java/C++
2. Three choices:
   - Field-insensitive: Merge all fields of each record object
   - Field-based: Merge each field of all record objects

- Field-sensitive: Keep each field of each (abstract) record object separate
  - Most precise



Records

Pointer Analysis Classification

1. Classify the pointer analysis algorithm we learned in this lesson:

Question	Answer
Flow-sensitive?	No
Context-sensitive?	No
Distinguishes fields of object?	Yes
Distinguishes elements of array?	No
What kind of heap abstraction?	Allocation-site based

Conclusion

1. What is pointer analysis?
   - Procedure for proving facts of the form pointer x and pointer y do not alias
2. May-alias analysis vs. must-alias analysis
   - May-alias is more useful in practice
3. Points-to graphs
   - Serves as an approximation of the state of data in the program
   - Determine if pointers may alias
4. Classifying pointer analyses:
   - Flow sensitivity
   - Context sensitivity

- Heap abstraction
- Aggregate modeling