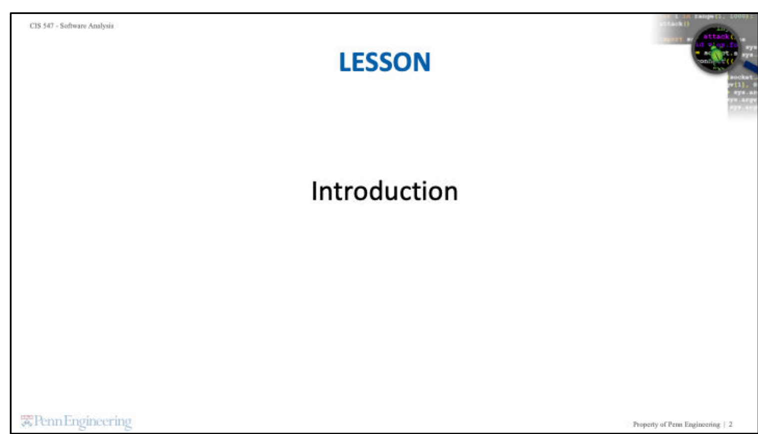
Previously, we learned how to reason about the flow of primitive data such as integers in a program. In this module, we will learn how to reason about the flow of non-primitive data, better known as pointers, objects, or references. This sort of analysis of a program is called a pointer analysis.

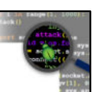
Pointers are prevalent in mainstream programming languages like C, C++, Java, and even Python. Therefore, pointer analysis is fundamental to any static analysis for reasoning about the flow of data in programs.

By the end of this module, you will have learned what pointer analysis is capable of doing and how you can incorporate it into your own dataflow analysis.



SEGMENT

May- and Must- Alias 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)
```


Let's begin with a dataflow analysis for an example program without pointers: just two integer variables, x and y. Notice that this program can be expressed in the WHILE language that we introduced in the previous module, once we extend that language to include the boolean equality operator. Suppose the goal of our dataflow analysis is to prove the assertion that y equals 1 at the end of the program.

To do this, we can perform a forward, must analysis that tracks the value of program variables. It begins by analyzing the assignment to x and infers that the value of x at this program point must be 1 ([x == 1] appears). It then analyzes the assignment to y, and infers that since the value of x before the assignment is 1, then the value of y after the assignment must also be 1 ([y==1] appears). The analysis thus proves that the assertion y==1 is valid at this point in the program.

Now let's consider a new example that is similar, but involves more complex types, in particular, pointers. Take a moment to read and understand the program on the right. We see three new types of statements here that are not covered by our previous definition of the WHILE language.

The first statement contains on object allocation using the keyword "new". As in C++ or Java, this statement allocates a portion of memory for a new object of type Circle. Specific sections of memory within that portion are reserved for each field of the object. The variable x is assigned to the location of the allocated section of memory.

In the next statement, we set the field "radius" to 1. More explicitly, we are accessing the portion of memory that x refers to and writing the integer 1 in the space dedicated to the "radius" field. Appropriately, we call this a "field write."

Conversely, in the next statement , “y = x.radius”, we perform a "field read." This is done by accessing the specific portion of memory dedicated to the radius field within the portion allocated for x. We read this value and set y equal to it.

Now that we understand what this program does, let's discuss how to analyze it.

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


We begin by analyzing the assignment to x.radius. We can infer that the value of x.radius at this program point must be 1 (write [x.radius == 1]). We then analyze this assignment to y, and we can infer that since the value of x.radius before the assignment is 1, the value of y after the assignment must also be 1 (write [y == 1]).

Our analysis therefore proves that this assertion is valid.

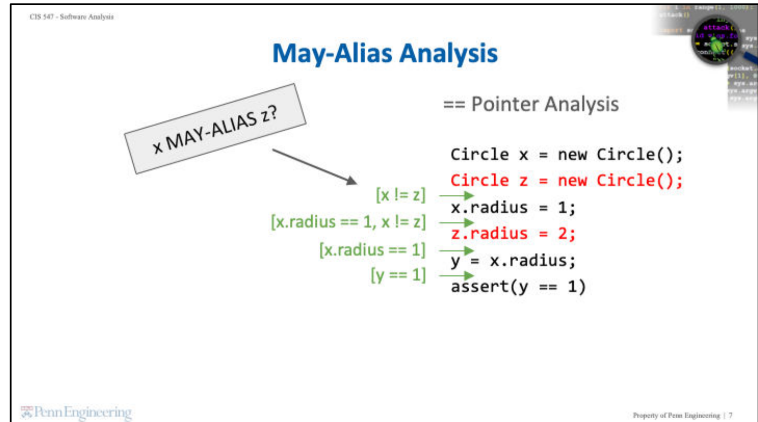
Notice that, this time, our analysis had to track the values of expressions more complex than variables, notably x.radius.

Pointer Aliasing

- Situation in which same address referred to in different ways

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

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

Expressions built using pointers, such as x.radius, allow the same memory address to be referred to in different ways. This situation is called pointer aliasing. In other words, pointer aliasing is when two expressions denote the same memory location.

The example we just looked at (gesture to example on left) did not have any pointer aliasing, since we had only one Circle pointer.

Let's look at a slightly different example that does have pointer aliasing and see what challenges it poses to our analysis (bring up example on right).

In this example, we have two Circle pointers, denoted x and z, but let's not commit yet to what z points to. Also note this additional assignment statement that writes 2 to the radius field of the Circle denoted by z.

Our analysis proceeds as before. After this assignment (point to statement x.radius = 1), we infer that the value of expression x.radius is 1. But after this assignment (point to statement z.radius = 2), our analysis is stuck: we do not know whether the value of expression x.radius should remain 1 or become 2. The answer depends on whether or not z is an alias of x.

Let's consider the two cases: one in which z denotes a different circle than x, and the other in which z denotes the same circle as x.

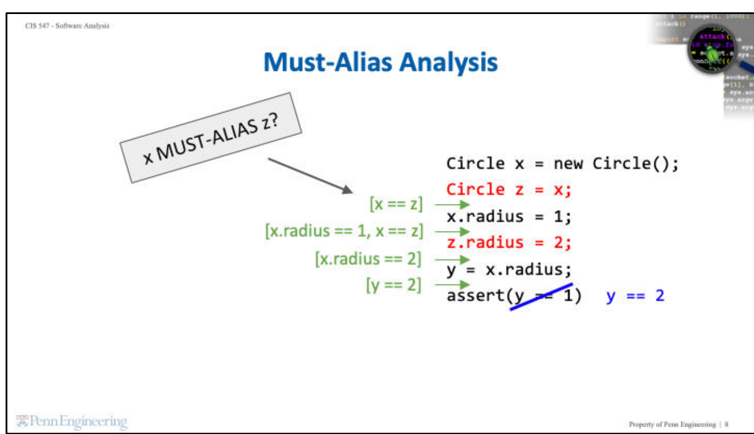
Here, let's suppose x and z denote different Circles (underline new Circle()).

In this case, our analysis proceeds as follows.

After this assignment to z, we infer that x and z denote different circles. Continuing further, after this assignment to x.radius, we infer that the value of x.radius is 1. Now we analyze the assignment to z.radius. This time, we conclude that the value of x.radius remains 1 after this assignment because we tracked the fact x != z. Finally, we inspect this assignment (point to assignment) to y, and we infer that, since the value of x.radius is 1 before the assignment, the value of y must be 1 after the assignment, thereby proving the assertion.

To recap, our analysis was able to prove this assertion by tracking the fact that it is NOT true that x and z may alias (box and arrow appear). An analysis that is dedicated to proving facts of this form is called a MAY-alias analysis. MAY-alias analysis is also what we call pointer analysis.

At this point, you might be wondering: just as we had MAY vs. MUST dataflow analyses, is there a counterpart to MAY-alias analysis? The answer is yes, and, as you might expect, it is called MUST-alias analysis.



To understand must-alias analysis, let's consider the alternative case in our example program, where x and z denote the same circle (underline x). In other words, x and z are aliased.

In this case, our analysis proceeds as follows.

After this assignment to z (point to statement), we infer that x and z denote the same circle. Continuing further, after this assignment to x.radius (point to statement), we infer that the value of x.radius is 1. And after analyzing the assignment to z.radius, we conclude that the value of x.radius becomes 2 after this assignment. We're able to conclude this fact because we tracked the fact that x and z must alias (box and arrow appear).

Finally, we look at the assignment to y. We infer that since the value of x.radius is 2 before the assignment, the value of y must be 2 after the assignment. The analysis thus fails to prove the assertion. Indeed, this assertion is incorrect. The correct assertion should be y == 2, which is what our analysis also proves (strike out y == 1 and hand-write y==2).

To recap, our analysis was able to prove this assertion by tracking the fact that x and z MUST alias.

CIS 547 - Software Analysis

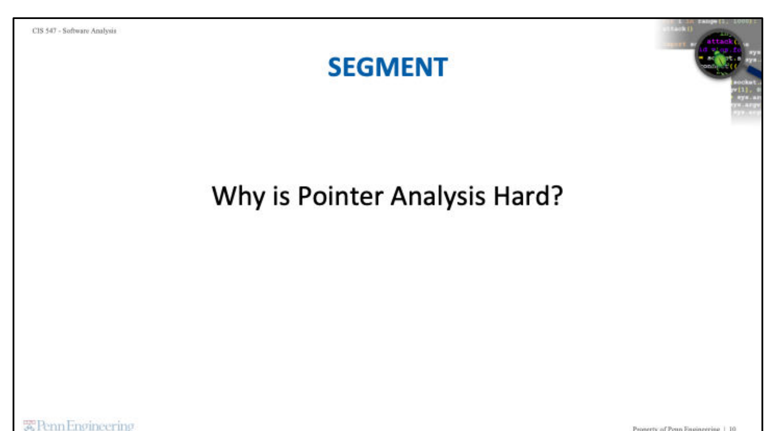
May-Alias vs. Must-Alias Analysis

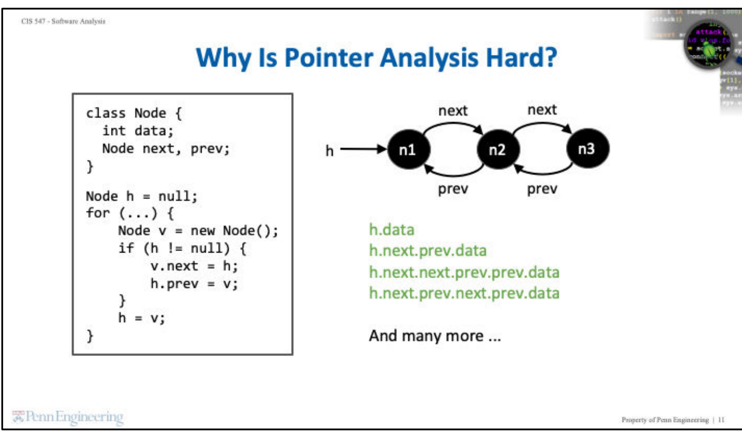
- May-Alias and Must-Alias are dual problems
- Must-Alias is more advanced and less useful in practice
- Main focus of this module: May-Alias Analysis

Penn Engineering

Property of Penn Engineering | 9

May-alias analysis and must-alias analysis are duals of each other. But the technical machinery needed for must-alias analysis is far more advanced than that for may-alias analysis. Also, may-alias analysis is useful for many more practical dataflow analysis problems than must-alias analysis. Therefore, in this lesson, we will focus on may-alias analysis, which is also called pointer analysis.





Before we dive into how to do pointer analysis, it is worthwhile to look at why we need a separate type of analysis for pointers. This will help make the motivation for the pointer analysis algorithm more clear.

Dataflow analysis in the presence of pointers is more challenging than dataflow analysis in the absence of pointers for a couple of reasons. Let's take a look at the following data structure, a doubly linked list, created by this example program (code snippet and graph appear). Each vertex is a Node object in memory, and each Node object has a data field and two pointer fields, next and prev, capable of pointing to other Nodes. A precise dataflow analysis of this program would need to keep track of all possible ways of accessing each Node's data.

For example, the data field of the Node denoted n1 could be referred to in many different ways. One way to refer to it is simply h.data. Another way to refer to it is h.next.prev.data. Yet more ways to refer to it are h.next.next.prev.prev.data, h.next.prev.next.prev.data, and so on.

Tracking all these different expressions is inefficient at best and infeasible at worst: in the presence of cycles, like in this example, there are infinitely many ways of referring to the same piece of data. It is clear that we need some kind of abstraction to keep track of aliasing information.

Approximation to the Rescue

- Pointer analysis problem is undecidable

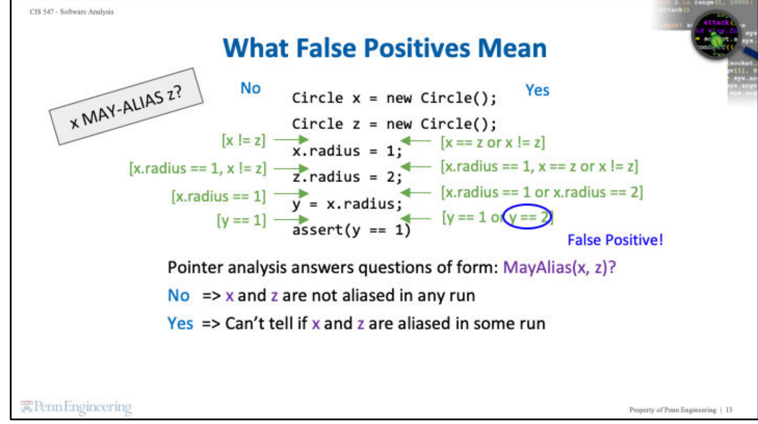
  => We must sacrifice some combination of:
  soundness, completeness, termination

- We are going to sacrifice completeness

  => False positives but no false negatives

Recall that dataflow analysis, in the absence of pointers, was undecidable. It follows that the problem of deciding whether two pointer alias is also an undecidable problem. In other words, there is no algorithm that always terminates and perfectly decides whether two pointers alias.

In order to create a feasible analysis, we must sacrifice either soundness, completeness, or termination. As we did for dataflow analysis, we decide to sacrifice completeness. This means, in exchange for the possibility of obtaining false positives, we can design an alias-detection algorithm that terminates and never gives false negatives.



It is worth being a bit more precise in what we mean by the term "false positive." Let's revisit our earlier example.

Remember that the question we are asking in this problem is: "Is it possible for two given pointers to be aliases of one another in some execution of this program?"

A shorthand version of this question is the boolean function "x MayAlias z". This function returns NO if there is no possibility that x and z are aliases, as is the case in this example.

Take a moment to convince yourself that this answer enables our dataflow analysis to eventually prove the assertion at the end of this program (bring up the green steps on the left).

Conversely, x MayAlias z returns YES if we cannot determine whether x and z are aliases or not. In other words, YES does not mean x and z MUST be aliases: it just means they may or may not be aliases. If x MayAlias z returns YES when x and z are not actually aliases, we consider this a "false positive."

Let's take a look at how a false positive manifests in this example. Suppose x MayAlias z returns Yes. Then, the most accurate information that our dataflow analysis can safely infer at this point is that x may or may not be equal to z. Continuing this reasoning, our dataflow analysis concludes at the end of the program

that the value of y may be either 1 or 2. The analysis thus fails to prove the assertion that the value of y must always be 1 at this point. The conclusion that the value of y may be 2 is a false positive whose existence can be traced back to the pointer analysis answering Yes to the question “x MayAlias z” when in fact x and z are not aliases.

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

There are many sound but approximate algorithms to the problem of pointer analysis.

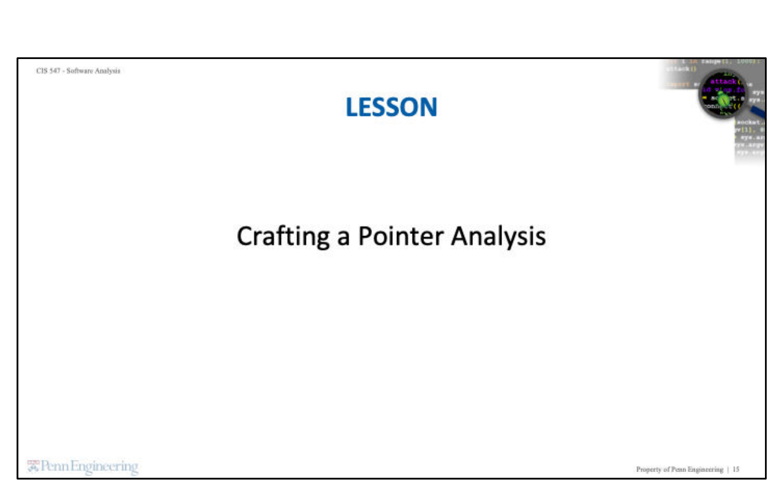
All these approximate algorithms generate false positives in certain circumstances, but they differ in their precision; that is, their false-positive rate.

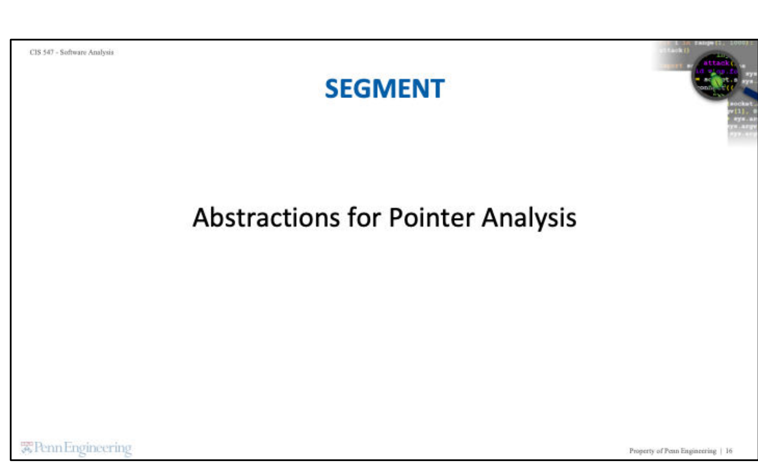
The approximations that these algorithms perform differ in two key aspects: how they abstract program data, in particular dynamically allocated data, which we will call the heap; and how they abstract control-flow.

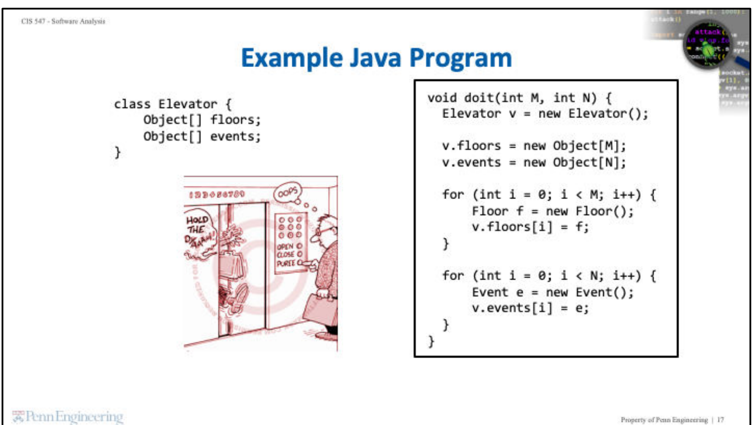
We already saw an abstraction of control-flow in the previous lesson on dataflow analysis where we approximated all branch conditions in the program’s control-flow graph using non-deterministic choice.

Pointer analyses typically go further in that they ignore control flow entirely and instead look at the program as a set of unordered statements. We call this a flow-insensitive analysis.

Let’s dive deeper into data and control-flow abstractions for pointer analyses using an example program.


CIS 547 - Software Analysis
LESSON
Crafting a Pointer Analysis
Penn Engineering
Property of Penn Engineering | 15


CIS 547 - Software Analysis
SEGMENT
Abstractions for Pointer Analysis
Penn Engineering
Property of Penn Engineering | 16



Throughout this lesson, we will use this Java program to illustrate the key concepts of pointer analysis. This program constructs a representation of an elevator.

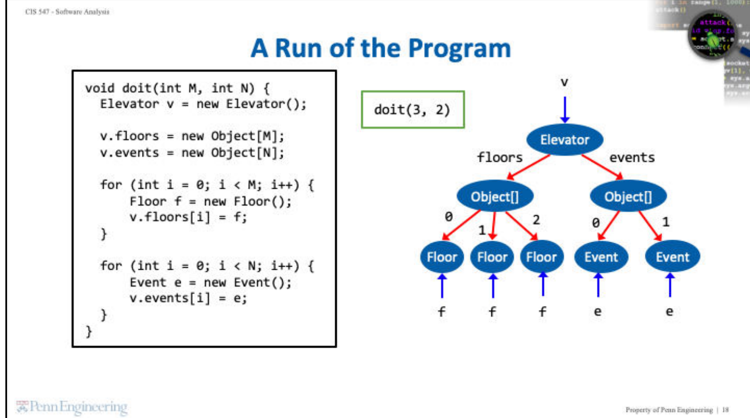
An Elevator object has two fields.

One field is an array of Floor objects (point to floors field) representing different floors in a building, such as the basement, 1st floor, 2nd floor, and so on.

The other field is an array of Event objects (point to events field). An example event is a person pushing a button for the 2nd floor in an elevator currently on the 5th floor.

The doit function takes as input the number of floors M and the number of events N. It starts out by creating an object of class Elevator. It then initializes the floors and events fields of the created Elevator object.

Let's take a look at a sample concrete run of this program.



In this run, we will create an elevator for a building with three floors and two events. That is, we will assume that the doit function is called with M = 3 and N = 2.

When the doit function is called, the allocation of "Elevator" is evaluated first. This allocation calls the Elevator's constructor method, which we do not show. Having completed the construction of the Elevator object, we assign the address of this Elevator object in memory to the variable v.

In the next line of the program, we first allocate an Object array with 3 cells, and then we write the location of this array in memory to the floors field of the Elevator object we're constructing.

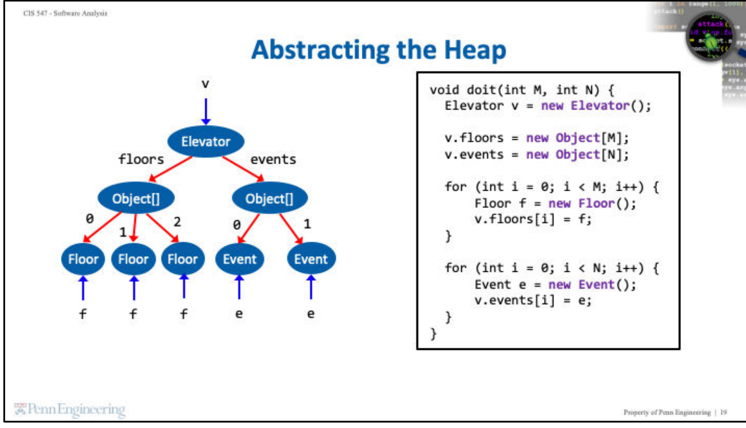
In the next line, we follow similarly, but now we're allocating an Object array with just 2 cells, and we're writing the location of this array to the events field of the Elevator object.

In the first iteration of the first for-loop (point at first for-loop), we take four steps. We allocate a new Floor object, assign the Floor object's memory address to the pointer f, read the floors field of the Elevator object being constructed, and do a field-write of the address in f to the 0th cell of the Object array at the address contained in the floors field. We then repeat this procedure for the second and third iterations of this loop.

Similarly, for the first iteration of the second for-loop (point at second for-loop), we

allocate a new Event object, assign the Event object's memory address to the pointer e, read the events field of the Elevator object, and do a field-write of the address in e to the 0th cell of the Object array at the address contained in the events field. We then repeat this procedure for the second iteration of this loop.

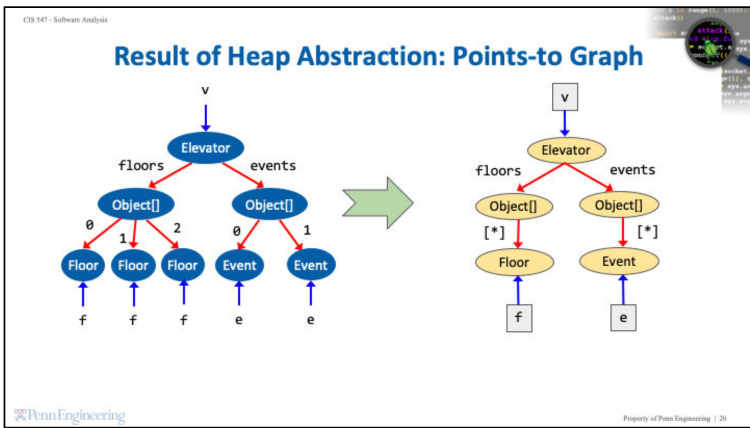
This concludes the operation of the doit function. The graph on the right represents the points-to relationships in this concrete run of the program.



This run of the elevator program created only three floors and two events, but a pointer analysis must be able to reason about each run with M floors and N events, for any value of M and N.

Pointer analysis achieves this by abstracting the heap. There are many possible schemes to abstract the heap, each of which strikes a different tradeoff between precision and efficiency. One of these schemes abstracts objects based on the site at which they are allocated in the program. We will look at other schemes later in this lesson.

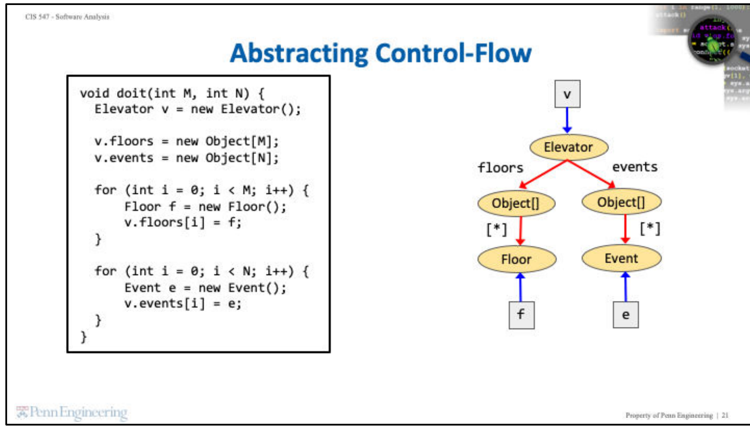
The elevator program has five allocation sites. Therefore, pointer analysis operates on the following graph, which conflates all objects allocated at the same site into a single node of the graph.



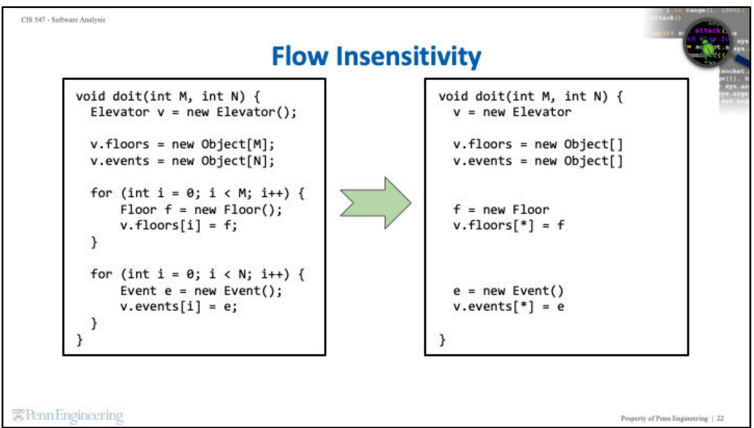
In particular, it collapses all the Floor nodes into a single Floor node, and all the Event nodes into a single Event node. Instead of labeling the pointers from the Object array nodes to these individual objects by array indices, we use the nondeterministic-choice symbol asterisk. And also observe that the variables f and e now point to a single node each instead of multiple nodes.

The type of graph that this heap abstraction produces is called a “points-to” graph. In a general points-to graph, there are two kinds of nodes: variables and allocation sites. Variables will be denoted by boxes, and allocation sites will be represented by ovals.

Since the points-to graph relation is an asymmetric relationship (that is, pointers between data need not go both ways), this is a directed graph. So we will represent edges with arrows. There are two types of edges in this graph. Arrows from a variable node to an allocation site node will be colored blue, and arrows from one allocation site node to another allocation site node will be colored red and labeled by a field name.



Although points-to graphs are finite, it is too expensive in practice to track a separate such graph at each program point. This is in contrast to the dataflow analyses we learnt in the last lesson where we tracked a separate set of dataflow facts at each program point. Instead, most pointer analyses only track a single, global points-to graph for the entire program. They achieve this by abstracting control-flow.



The particular control-flow abstraction that is commonly used by pointer analyses is called flow insensitivity. Applying this abstraction to our example elevator program produces the following abstract program.

There are three major differences I'd like to highlight here:

1) Notice that all control-flow features have been removed, including constructs like for-loops and also semicolons indicating sequentially ordered statements.

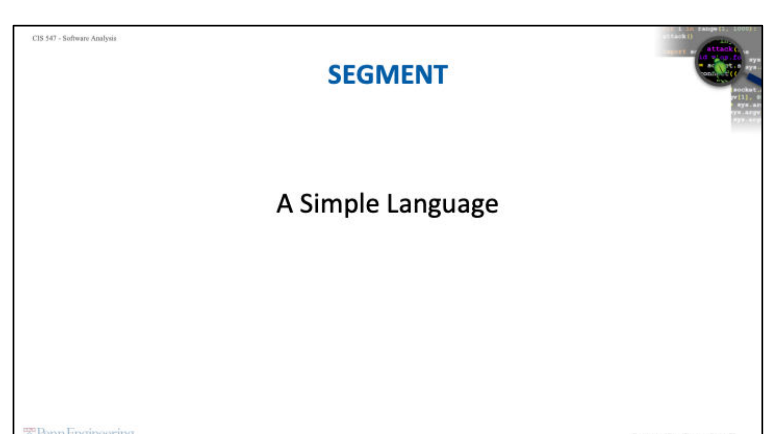
2) All statements that do not affect pointers have also been removed. For example, the approximated code no longer has statements setting the integer i equal to 0 and incrementing i.

3) Finally, array indices are replaced by nondeterministic choice, denoted by the asterisk symbol. This is similar to how conditions at branch points in dataflow analysis were replaced by nondeterministic choice.

This abstraction can be thought of as turning the program into an unordered set of statements. Even though we've written the statements in the rough order they appear in the original program, it is better to think about the statements as having no precedence over each other.

Even though this abstraction appears to lose a lot of information, we will see later in this lesson that it still able to prove interesting properties, such as the property that variables e and f do not alias.

Next, let's see how a pointer analysis algorithm builds a points-to graph for an arbitrary program.



Kinds of Statements

(statement) s ::= v = new … | v1 = v2 | v1 = v2.f
| v1.f = v2 | v1 = v2[*] | v1[*] = v2

(pointer-typed variable) v

(pointer-typed field) f

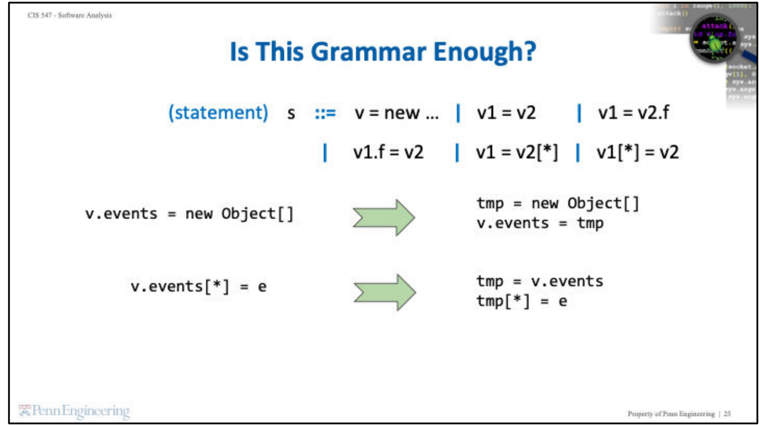
For pointer analysis, it suffices to consider the following kinds of statements. The syntax here is similar to a Java program.
(point to each statement in turn)

First, we must consider object allocation statements which create pointers using the new keyword.

In addition to pointer creation, we also must consider when a variable aliases that pointer. This situation is shown by the assignment v = v2. We call this an "object–copy" statement because the memory address referring to an object is copied from v2 to v.

Next, we must consider field-reads and field-writes. These statements use the dot operator to access fields on an object.

Lastly, we must consider the case in which array members alias pointers. In other words, when an array holds pointers to an object. We consider both cases in which we read and write to these members.



At this point, you might be wondering whether the grammar we just presented is sufficient to represent all the operations that we might need to reason about in order to determine whether two pointers may alias. The answer is yes.

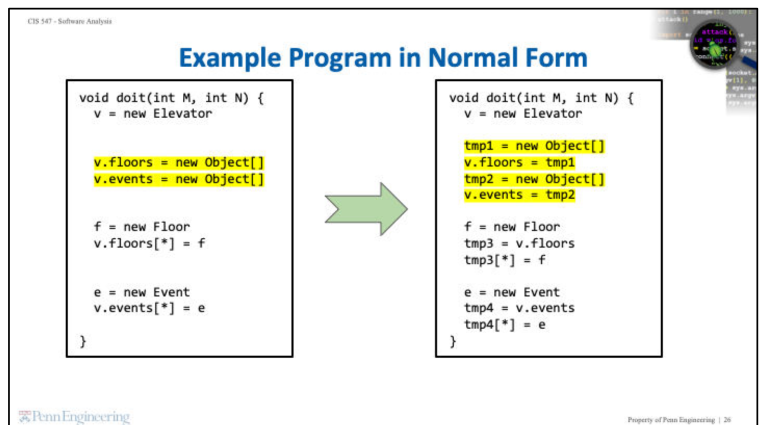
We will not give a formal proof of this fact in this lecture. However, you can convince yourself this is sufficient by the intuition that all of the ways in which aliasing can occur are covered by this grammar.

We will now go through some examples of how to break down more complicated statements into compositions of these six simpler types of statements.

First, let's consider the statement "v.events = new Object[]" from our elevator program. This statement starts with an allocation of the Object array, and then it assigns the address of the new allocation to the field events of the Elevator object. We can break this statement down into a field write and an object allocation.

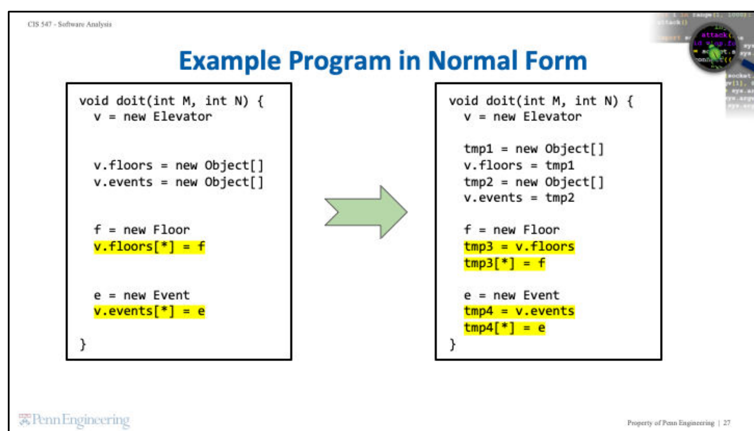
Now, let's consider our second example, "v.events[*] = e". It performs a read of the events field of the Elevator object, and then it assigns the content of e to some cell in the Object array that v.events points to. We can again break this statement up into two separate statements of the forms above. A field read occurs accessing the events field on object v. We also have an array write statement where the value of v.evennts[*] is set.

It is now clear that complex statements involving pointer aliasing can be broken into two simple statements of the form specified in our grammar.



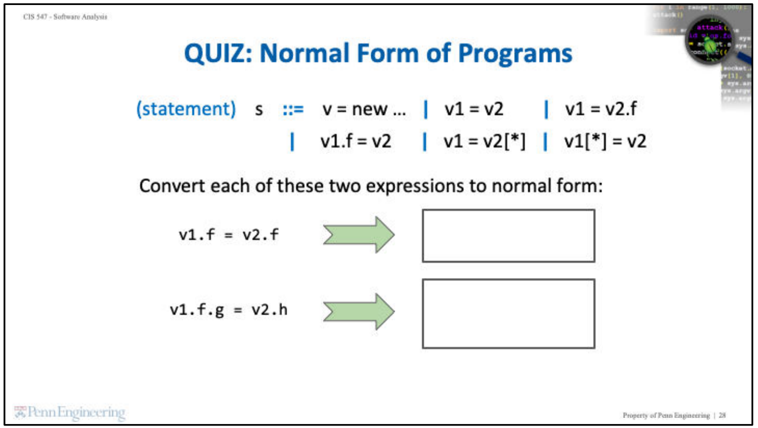
Let's decompose the statements in our Elevator program.

First, we decompose the highlighted statements. These statements contain both a field write and an object allocation statement. By adding a tmp variable, we decompose these statements into two statements, one for each update rule.



Now, let's decompose the highlighted statements. These statements contain two field-reads. By adding a tmp variable, we decompose the single statement into two field-read statements.

Now that the program is in normal form, we are ready to create a points-to-graph.

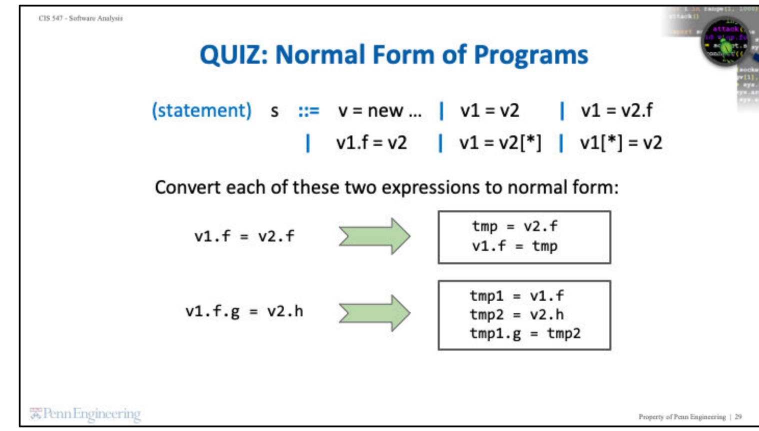


{QUIZ SLIDE}

Let's do a quick quiz to check your understanding of the form of programs that we'll be using for pointer analysis, which we will call a normal form.

For each of these two statements, "v1.f = v2.f" and "v1.f.g = v2.h" fill in the box on the right with equivalent statements in the normal form specified by this grammar.

Note that you may need more than two simple statements to capture the entire compound statement.



{SOLUTION SLIDE}

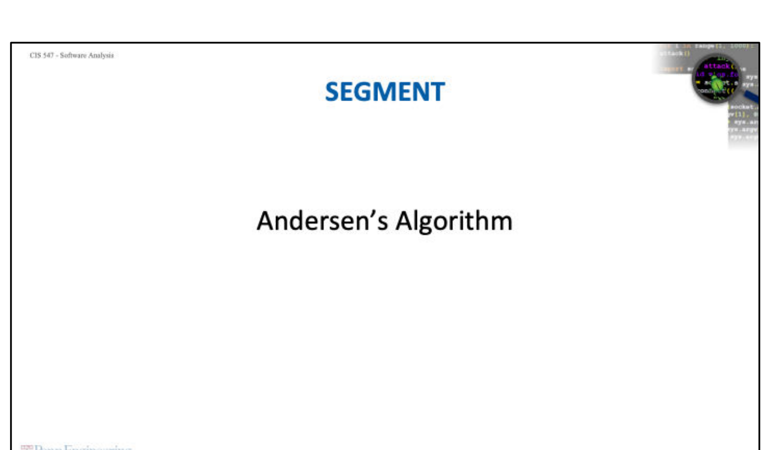
Let's look at some example solutions. Your answers might not look exactly the same, but they should have the same patterns.

For the first statement, v1.f = v2.f, we first read the f field of v2, and then we write the contents of v2.f to the field f of v1. Therefore, we need both a field-read statement and a field-write statement to capture this entire operation. I'll first assign the contents of v2.f to tmp (tmp = v2.f appears), and then I'll assign the contents of tmp to v1.f (v1.f = tmp appears).

For the second statement, v1.f.g = v2.h, we'll need more than two simple statements. We need to access the contents of v1.f in order to reach g, so that will be the field-read operation: tmp1 = v1.f (tmp1 = v1.f appears).

We also need to access the contents of v2.h, so that will be another field-read operation: tmp2 = v2.h (tmp2 = v2.h appears).

Finally, we need to write the contents of v2.h (in tmp2) to the field g of v1.f (in tmp1). We use a field-write operation to do so: tmp1.g = tmp2 (tmp1.g = tmp2 appears).



Andersen's Algorithm

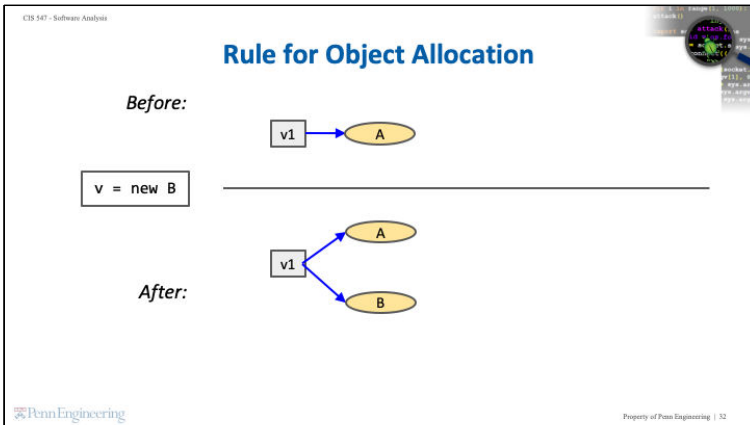
```
graph = empty
repeat:
    for (each statement s in program)
        apply rule corresponding to s on graph
until graph stops changing
```


One of the most well-known pointer analysis algorithms is Andersen's algorithm. The algorithm, that builds the points-to graph for a given program, was published in 1994.

Note that the structure of the algorithm is similar to the Worklist algorithms that we presented in the dataflow analysis module.

Starting with an empty graph, we examine statements that create pointers and update the graph based on points-to-relationships. The algorithm terminates when the graph stops changing.

The set of statements that we iterate over is obtained using the flow-insensitive abstraction we just presented. In addition to removing control flow, we also removed any statements that did not "affect pointers." Let's dive into the details of which statements are included.



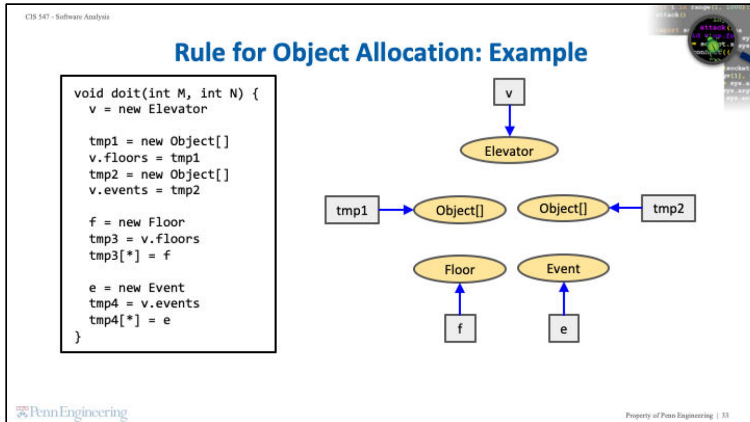
By now, you should be convinced that we can take a program written in our simplified Java-like language and express its pointer operations using the six simple statements we introduced earlier.

In order to perform Andersen's algorithm, we need to understand how each type of statement affects the points-to-graph. The graph is updated depending on the type of statement currently being analyzed.

To make things more concrete, let's see how an object allocation statement affects points to relations.

For a statement v = new B, we create a new allocation site node called B, we create a variable node for v (if it doesn't already exist), and then add a blue arrow from the variable node to the allocation site node.

Note that if there is already an arrow from v to another allocation site node (say A), we just need to add a new arrow from v to B. This rule, as well as all the remaining rules we'll discuss, is a "weak update," in which we accumulate instead of replace the points-to information (which would be a "strong update"). This type of update rule is a hallmark of flow-insensitivity.



Looking at our example Elevator program, let's use the object allocation sites rule to create a partial points-to-graph.

The first statement contains an allocation statement that creates an object of type Elevator. Accordingly, we create a variable node for v, an allocation site node for the Elevator object, and connect them with a directed arrow. Abstractly, this represents, "v points to Elevator"

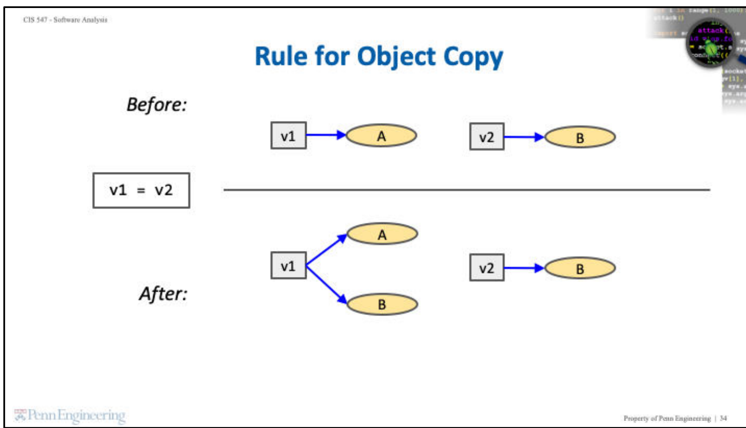
Similarly, we have four more allocation statements in this program:

tmp1 points to a new Object array (point to "tmp1 = new Object[]" as appropriate nodes and arrow appear on right-hand side),
tmp2 also points to a new Object array (point to "tmp2 = new Object[]" as appropriate nodes and arrow appear on right-hand side),
f points to a new Floor object (point to "f = new Floor" as appropriate nodes and arrow appear on right-hand side), and
e points to a new Event object (point to "e = new Event" as appropriate nodes and arrow appear on right-hand side).

Notice that we create separate nodes for the two Object arrays as they are allocated at different sites.

We now have a simple points-to-graph that represents the relationships between newly

allocated objects and the variables they were initially assigned to.



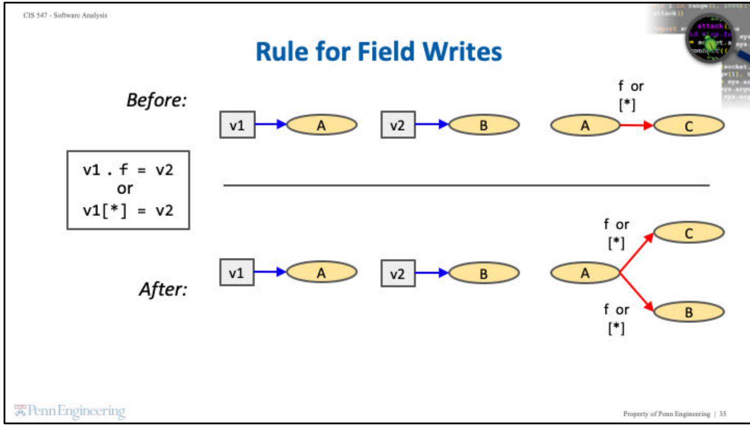
Now, let's expand our points-to-graph to include a few more relationships.

First, let's add alias relationships created through "object copy."

For the statement v1 = v2, we create a variable node for v1 (if it doesn't already exist), and then add a blue arrow from the variable node for v1 to all nodes pointed to by the variable node for v2.

Again note that we do not remove or replace any existing arrows from v1, such as this one (gesture). v1 merely accumulates another arrow to B.

In words, v1 now additionally points to what v2 points to.



Now, let's see how field writes affect relationship in a points-to-graph. Field-write statements can be one of the following forms (gesture to the two forms).

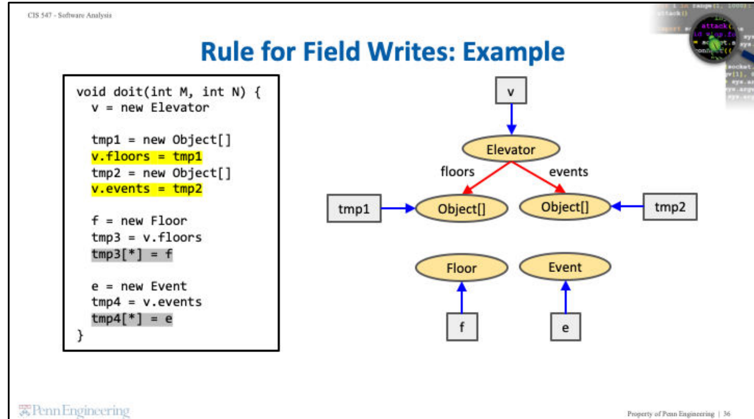
Although the two statements shown here are syntactically different, they will have the same affect on a points-to graph.

Prior to the field write statement, suppose v1 points to A and v2 points to B. Additionally, A has a link between its allocation site and the allocation site for the field f, or equivalently, the array member denoted by [*]. This represents the portion of memory specifically reserved for the field or array member.

The field write statement modifies the points-to-graph by adding a field relationship, denoted by a red arrow, from A to the value stored in v2. That is, we add a red arrow from A to B. Again, we perform a "weak update" and do not remove the field relationship between A and C.

Now, there are some edge cases we must consider. For instance, what should we do if v1 and v2 point to the same node? This would be equivalent to the statement v1.f = v1. As you might have guessed, we will add a red arrow from A to itself.

As another edge case, what should we do if there isn't already a node for v1 or v2? In this case, we should skip this statement temporarily and handle it in the next iteration.



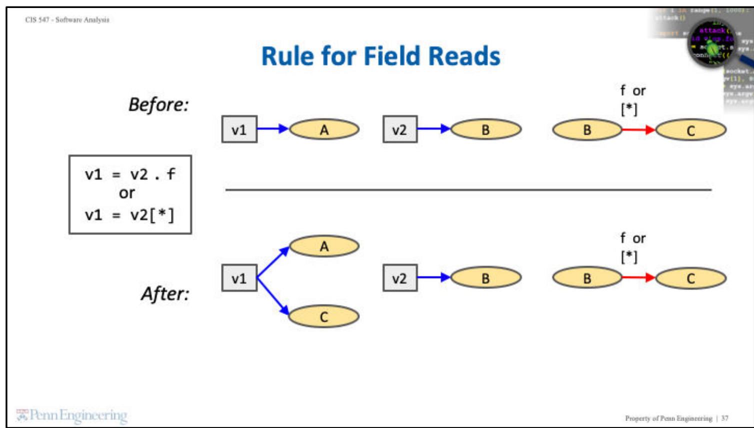
Now let's update our points-to-graph for the elevator program by applying the field-write update rule.

Shown on the right is the partial points-to-graph created by applying the object allocation update rule.

In this first field-write highlighted here, v.floors is set to tmp1. By the rule we just discussed, we add a red arrow from the allocation site v points to, to the allocation site that tmp1 points to. That is, a red arrow representing the "floors" field is added from Elevator to the Object array pointed to by tmp1.

Likewise, in the second highlighted field write, we add a red arrow representing the "events" field from the Elevator allocation site to the Object array pointed to by tmp2.

The latter two field-writes in this program are through arrays. However, since the variables nodes for tmp3 and tmp4 haven't been created yet, we skip these field-write operations and will come back to them in a later iteration.



Now, let's see how field-read relationships update a points-to-graph. As in field-writes, field-reads also have an equivalent rule update for reads from array members.

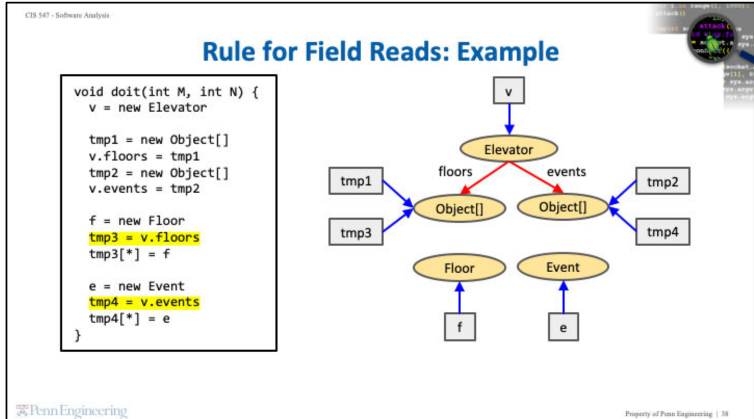
Suppose we have the setup shown here prior to the field-read statement. That is, v1 points to the allocation site A, V2 points to the allocation site B, and B has a link to C denoting field f, or, equivalently, array access [*].

Field-reads, in themselves, do not modify any points-to relationships. However, an assignment to a field read, can indeed modify relationships and thus will have an affect on our points-to-graph.

Following the shown field-read assignment, the variable v1 now points to the allocation site for field f (or array member [*]) of the object pointed to by v2. So, we add a arrow from v1 to C by following the arrows representing field relationships.

As in all update rules, we perform a "weak update" and keep the points to relationship between v1 and A. Note that object B may in fact be pointing to many other objects via arrows labeled by the field in question. In this case we would need to add an arrow from v1 to each of these nodes in order to reflect the fact that the field f, and therefore v1, may point to any one of these objects.

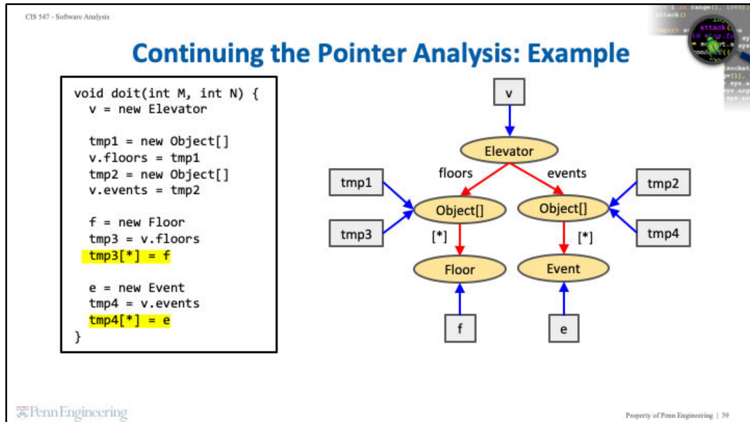
Lastly, we must consider the case in which the variable nodes for v1 or v2 do not exist. As you might guess, we skip the statement temporarily and try to handle it in the next iteration. Likewise, if there is no field relationship from B via f or [*], we skip the statement for this iteration.



Now let's extend our points-to-graph for the elevator program by applying the field-read update rule.

In this first field-read, we create a node for the variable tmp3. We add a blue arrow from tmp3 to the Object[] allocation site by following the arrows from v via the "floors" field relationship.

Similarly, we create a node for the variable tmp4, and add a blue arrow from tmp4 to the Object array which the "events" field of the Elevator points to.

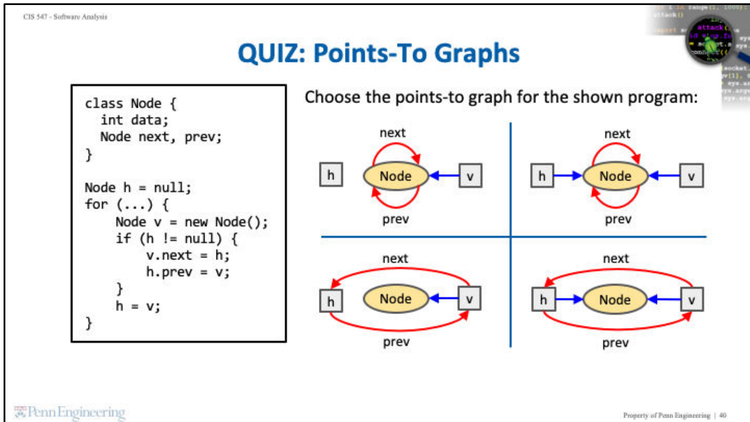


Now that we've created nodes for tmp3 and tmp4, we can apply the field-write rules to the statements we skipped previously.

Let us first consider the field-write statement tmp3[*] = f. By the field-write update rule, we add a red arrow labeled by an asterisk from the Object array pointed to by the floors field to the Floor node.

Similarly, for the next field-write statement tmp4[*] = e, we add a red arrow labeled by an asterisk from the Object array pointed at by tmp4 to the Event node pointed at by e.

At this point, iterating through the set of statements again will produce no changes to the graph, so the algorithm will terminate, leaving us with the points-to graph we see here.

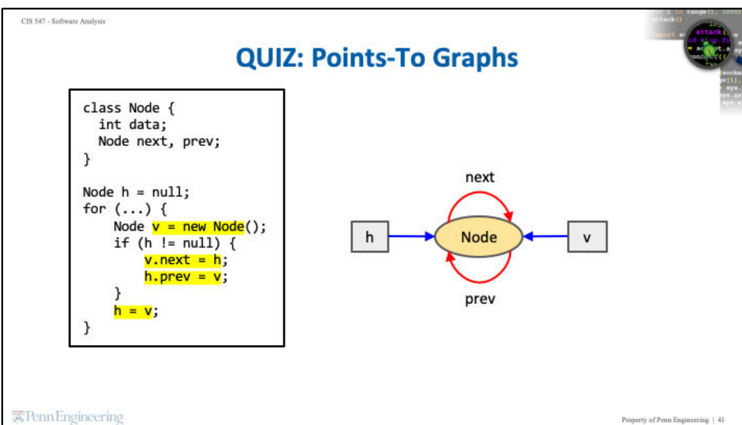


{QUIZ SLIDE}

Now that we've looked at all the rules for pointer analysis, let's return to an example we saw at the beginning of the module. Here we have some code for creating a doubly linked list of Node objects.

For this quiz, transform the code on the left-hand side using the flow-insensitivity approximation scheme, and then perform the algorithm yourself on the resulting set of statements. What does the points-to graph look like when the Andersen's algorithm concludes?

Click the radio button corresponding to the graph that represents the correct points-to graph.



{SOLUTION SLIDE}

The answer to this quiz is the top-right graph.

Let's work through the example together and see how the Andersen's algorithm gets us here.

First, we transform the code by applying the flow-insensitivity approximation, which results in the following four highlighted statements (highlight relevant statements from code). Note that we didn't highlight the statement "h = null". Remember that pointer analysis is a weak update analysis, so we never remove an edge from the graph once we add it in. Instead, the statement "h = null" just tells us to add no new edges to the node for variable h. So we can safely ignore this statement altogether in our analysis.

Now let's proceed through the highlighted statements, one statement at a time, and build the points-to graph iteratively.

The first statement we will look at is "v = new Node". Applying the rule for object allocation, we create a variable node for v, an allocation site node called "Node", and a blue arrow from v to "Node" (Node, v, and blue arrow from v to Node appear).

Let's now look at the statement "h = v" (remember that we can iterate through the

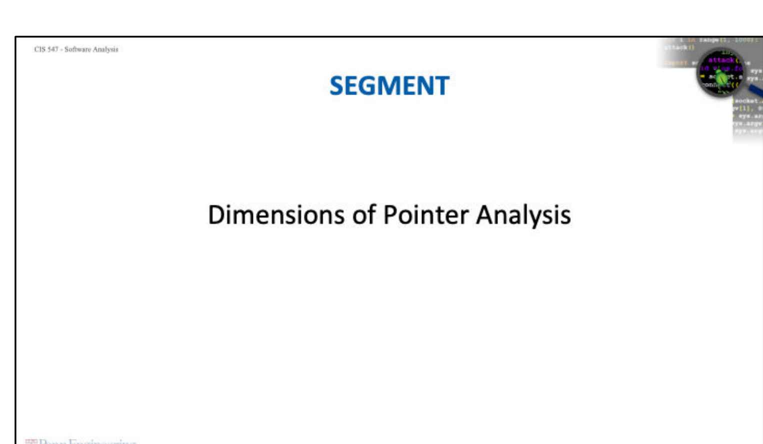
statements in any order, since the statements are coming from an unordered set). Applying the rule for this statement, we create a variable node for h and a blue arrow from h to the Node object that v points to (h and blue arrow from h to Node appear).

Now let's apply the rule for the field-write statement "v.next = h". Since both h and v point to the Node object, we create a red edge from the Node object to itself and label it "next" (add red arrow from Node to itself labeled "next"). Similarly, for the field-write statement "h.prev = v", we create a red edge from the Node object to itself and label it "prev" (add red arrow from Node to itself labeled "prev").

Iterating through the set of statements again doesn't cause us to change the graph at all, so the algorithm terminates, leaving us with the graph we see here as our points-to graph.

LESSON

Pointer Analysis Design Choices



Classifying Pointer Analysis Algorithms

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

Andersen's algorithm is one example of a pointer analysis. As with other analyses, points-to analysis has many different "knobs" that we can tune for increasing precision or efficiency. However, an increase in precision usually come at a cost in efficiency.

We can classify a points-to analysis algorithm across many different dimensions. Each dimensions allows the reader to understand, at a high level, how the algorithm balances precision and efficiency.

In order to classify a given analysis across four dimensions, we can ask the following questions:
1) Does the analysis consider control flow?
2) Does the analysis analyze a single function or an entire program at once?
3) How does the analysis abstract the heap?
4) How are aggregate data types such as records or classes modeled?

Let's discuss each of these four dimensions more deeply.

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

Flow-sensitivity concerns how a pointer analysis algorithm models control-flow within a procedure or function. As we are only concerned with control flow within each function, we call this *intra*-procedural control flow. We can classify an analysis based on how it models control flow within a function as flow-sensitive or flow-sensitive.

Flow-insensitive pointer analysis algorithms, like the one we just discussed, ignore control-flow entirely, viewing the program as an unordered set of statements. A hallmark of flow-insensitive analyses is that these analyses only generate new facts as they progress; they never kill any previously generated facts. We observed this in the case of the pointer analysis algorithm we just saw, wherein the points-to graph only grew in size as each statement of the program was considered. We say that such algorithms perform *weak updates*. Such algorithms typically suffice for may-alias analysis, that is, it is practical for a may-alias analysis to have a low false positive rate despite being flow-insensitive.

Flow-sensitive pointer analysis algorithms, on the other hand, are capable of killing facts in addition to generating facts. We say that such algorithms perform *strong updates*. This is possible as we have a distinct program order in which facts that come later take precedence over facts generated earlier in the program order. Such algorithms are typically required for must-alias analysis, that is, it is impractical for a must-alias analysis to have a low false positive rate by being flow-insensitive.

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

Another common dimension for classifying pointer analysis algorithms is context sensitivity, which concerns how to handle control-flow across procedures. This property is called *inter*-procedural control-flow. We can classify pointer analysis algorithms as context-insensitive or context-sensitive, based on how they handle inter-procedural control-flow.

Context-insensitive pointer analysis algorithms analyze each procedure once, regardless of how many different parts of the program call that procedure. These algorithms are relatively imprecise, as they conflate together aliasing facts that arise from different calling contexts. But they are very efficient, since they analyze each procedure only once.

Context-sensitive pointer analysis algorithms, on the other hand, potentially analyze each procedure multiple times, once per abstract calling context. These algorithms are relatively precise but expensive. They differ primarily in the manner in which they abstract the calling context. Like many choices in analysis, the choice of the scheme is dictated by the desired tradeoff between precision and efficiency.

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

In addition to abstractions in control flow, pointer analyses may also have various levels in abstraction of dynamic memory. A heap abstraction concerns abstracting program data, in particular, dynamically allocated objects on the heap.

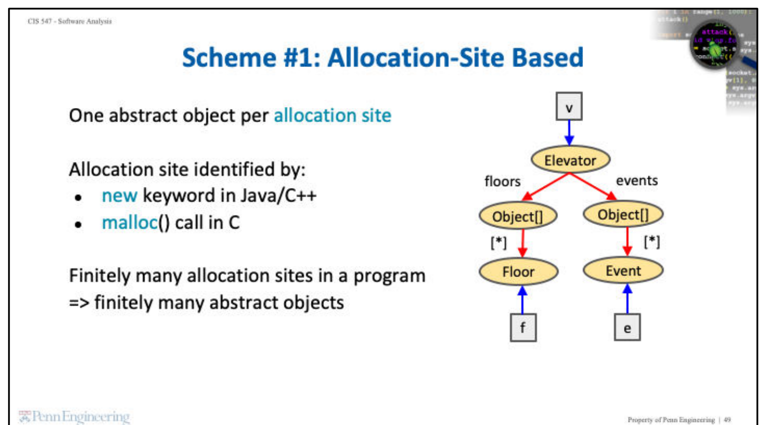
As we saw earlier in the lesson, keeping track of aliasing without abstractions can be expensive or even infeasible. In order to lessen this burden, analyses often create an abstraction of the dynamic allocation sites. A heap abstraction scheme specifies how to map a potentially infinite number of concrete dynamically allocated objects to a finite number of abstract objects. Each abstract object corresponds to an oval node in the points-to graph. This partitioning is at the heart of ensuring that pointer analysis terminates.

Much like any of the abstractions we have seen in this course, designing a suitable heap abstraction is an art. Many sound heap abstraction schemes exist with varying precision and efficiency. As a rule of thumb, a scheme that produces too few abstract objects in the points-to graph will result in an efficient but imprecise pointer analysis -- imprecise because it conflates more concrete objects into each of the few abstract objects. On the other hand, a scheme that produces too many abstract objects in the points-to graph will result in an expensive but precise pointer analysis.

Let's take a look at three heap abstraction schemes that are commonly used from most precise to least precise.

SEGMENT

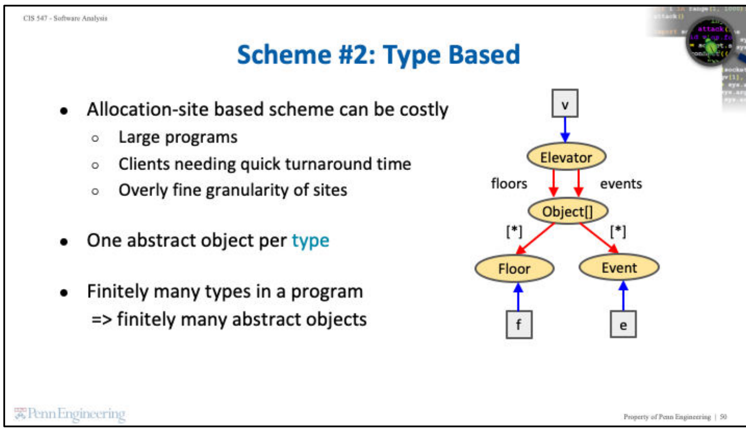
Heap Abstraction Schemes



The heap abstraction scheme used by Andersen's algorithm that we illustrated in this lesson is called the allocation-site based scheme. This scheme abstracts concrete objects based on the site in the program where they are created, called the allocation site. In other words, each abstract object under this scheme corresponds to a separate allocation site.

Allocation sites are identified by the new keyword in Java and C++, and by the malloc function call in C. Since there are finitely many allocation sites in a program, a pointer analysis using this scheme is guaranteed to result in a finite number of abstract objects in the constructed points-to graph.

Here is the points-to graph that was constructed using the allocation-site based scheme for our example elevator program. Each of the five ovals correspond to a separate allocation site in the program.



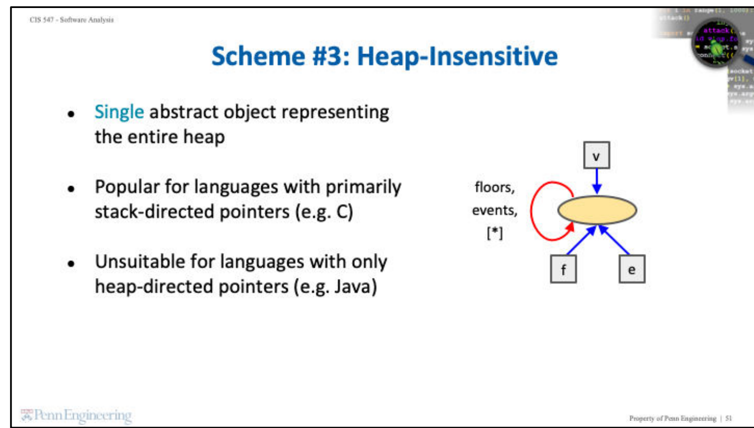
Now let's consider a slightly less precise scheme based on types.

Although the number of allocation sites in any program is finite, tracking a separate abstract object per allocation site can be prohibitively expensive for large programs, which can contain too many allocation sites.

This scheme partitions concrete objects based on their type instead of based on the site where they are created. In other words, each abstract object under this scheme corresponds to a separate type. Since there are finitely many types in a program, a pointer analysis using this scheme is guaranteed to result in a finite number of abstract objects in the constructed points-to graph.

Here is the points-to graph that would be computed for our example elevator program by a pointer analysis using the type-based scheme. Notice that each of the four ovals corresponds to a separate type in the program. In particular, notice that the two separate ovals that were created for the arrays of floors and events are now conflated into one, as both have the same type: an array of objects.

This scheme may be appealing to clients of pointer analysis that need a quick turnaround time to aliasing queries, such as an integrated development environment.



Now let's consider an even more imprecise abstraction scheme.

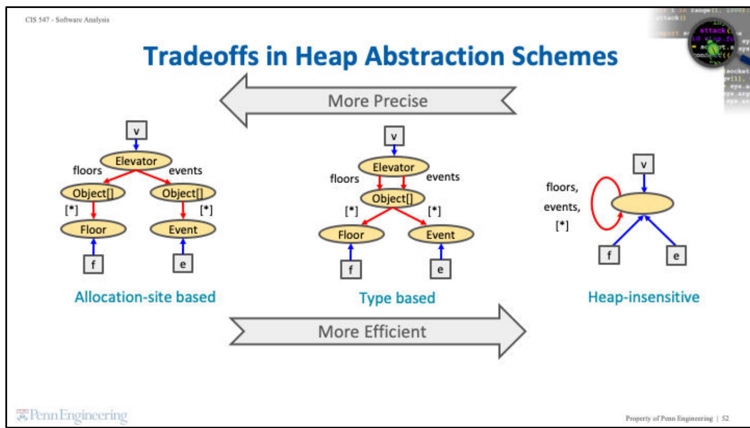
In this scheme, we do not make any distinctions between dynamically allocated objects. We call this "heap-insensitive". This scheme uses a single abstract object to model the entire heap.

The points-to-graph for our example elevator program is shown here. It contains a single abstract node for the entire heap. Looking at this points-to graph, it should be easy to see that this scheme is highly imprecise for reasoning about the heap, but it is sound nevertheless.

So you might wonder: are there scenarios in which this scheme is useful? The answer is yes, for languages with primarily stack-directed pointers like C, where malloc calls are sparse. Pointer analyses for such languages derive most of their precision by reasoning about stack-directed pointers, and completely ignoring heap-directed pointers.

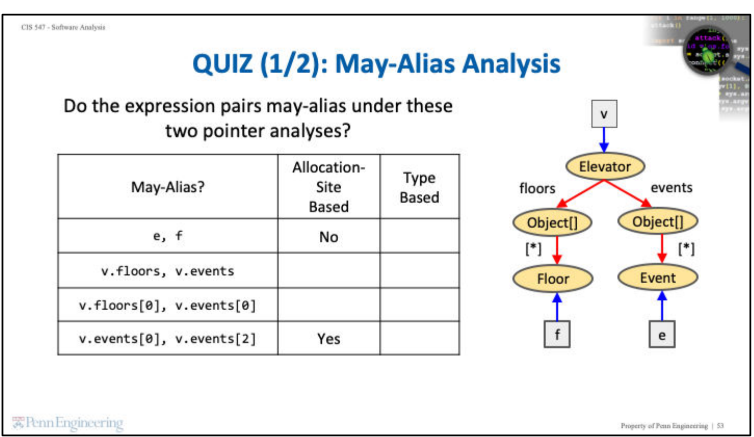
Of course, this scheme is unsuitable for languages with only heap-directed pointers like Java.

We will look at pointer analyses for programs with stack-directed pointers later in this module.



To recap, the three different heap abstraction schemes that we just saw strike a different tradeoff between precision and efficiency. As we go from the allocation-site based scheme to the type-based scheme to the heap-insensitive scheme, the pointer analysis gets more efficient, but also less precise.

It is important to remember that these are not the only three schemes: there are many other schemes in the literature, and you can even design your own depending on your analysis needs. For instance, there are many schemes that are more expensive and precise than the allocation-site based scheme; these schemes can even make distinctions between objects created at the same allocation site. At the same time, remember that we can never have a scheme that will allow pointer analysis to make distinctions between all concrete objects and terminate in finite time.



{QUIZ SLIDE}

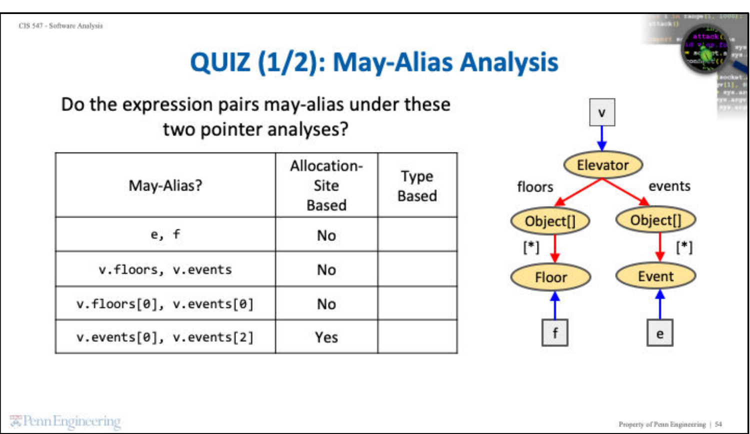
In order to better understand the tradeoff between precision and efficiency, let us calculate alias relationships between program variables. In particular, we will evaluate mayAlias relationships under the allocation site and type based schemes to note the differences.

Let us first calculate relationships based on the allocation site based scheme. The points to graph we generated using Andersen's algorithm is shown here.

If we ask the question, “MAY the pointers e and f alias?”, we can compute the answer by following the arrows in our points-to-graph. The answer is NO as the arrows from e and f refer to different abstract nodes in our points to graph. Thus e and f cannot be aliases.

On the other hand, if we ask whether v.events[0] and v.events[2] may-alias, the answer is YES. By following the arrow from v, to the Elevator node, followed by the red "events" field arrow, we reach the Object array node. Now, regardless of the index, accessing a member of events brings us to the Event node. That is, v.events[2] and v.events[0] are represented by the same abstract node, so, it is possible that they alias. Note that the concrete objects could be different, but we cannot say one way or the other using this heap abstraction scheme.

Now, fill in the remaining boxes in this table, answering the question “MAY the two given pointers alias?” under the allocation site-based heap abstraction.

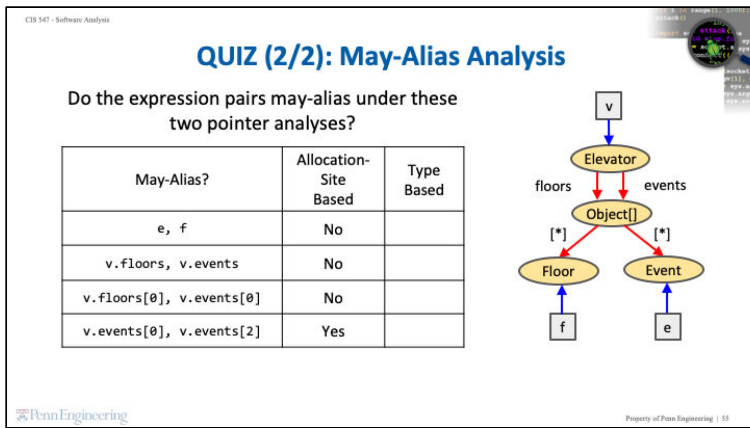


{SOLUTION SLIDE 1}

Let's work through the answers together to see why both of the following cannot alias.

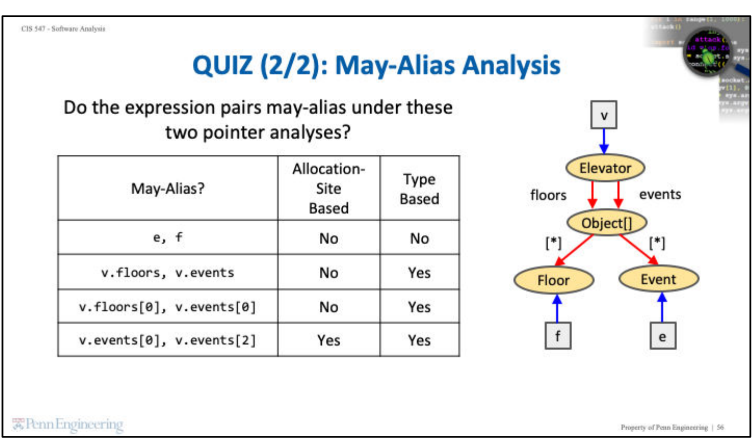
First, let's answer the question, mayv.floors and v.events alias? For v.floors, we follow the arrow from v to Elevator and then the arrow labeled "floors" to this (gesture) Object array node. For v.events, we follow the arrow from v to Elevator and then the arrow labeled "events" to this (gesture) Object array node. Since these two nodes are distinct in the points-to-graph, v.floors and v.events always refer to concrete objects allocated at different sites in the program. Thus they cannot refer to the same concrete object, so the question "MAY v.floors and v.events alias?" is answered "NO"

For v.floors[0] and v.events[0], we need to follow one more arrow each than we did when analyzing v.floors and v.events. We follow the arrows labeled by the asterisks. Since we end up at the Floor node for v.floors[0] and the Event node for v.events[0], we can again conclude that v.floors[0] and v.events[0] always refer to different concrete objects. So the question "MAY they alias?" is answered "NO"



{QUIZ SLIDE}

Now let's look at the points-to graph the algorithm generates for the type-based heap abstraction. Take a moment to fill in the whether each of the following May-Alias.



{SOLUTION SLIDE 2}

As before, we can answer the question, "may x and y alias?" By following the arrows in the points-to-graph.

As before in the allocation site-based abstraction, e and f point to two different nodes. This means that we can be certain e and f point to objects of a different type, so we can answer the question “MAY e and f alias?” by “NO”

However, we see a difference in the precision of these two analyses when asking whether v.floors and v.events may alias. Following the arrows from v to the Elevator node and then the “floors” arrow from the Elevator node, we reach the Object array node where v.floors points. But if we’d taken the “events” arrow from Elevator, we’d end up at the same Object array node. So the type-based heap abstraction scheme cannot distinguish that these two pointers don’t actually alias in our example program. It answers “YES” to the question of whether they MAY alias.

Now, suppose we wanted to know whether v.floors[0] and v.events[0] MAY alias. We first follow the arrows to the Object array node that v.floors and v.events lead us to. Now we need to follow the arrow labeled by an asterisk, but there are two such arrows leading away from the node. We need to follow both in parallel: all this graph tells us is that v.floors[0] could point to either the Floor node or the Event node. And it tells us the same thing for v.events[0]: this pointer could refer to either the Floor node or the Event node. Since the set of nodes that v.floors[0] could point to and the set of nodes that v.events[0] could point to overlap, our analysis cannot prove that they do not point to the same object. So we answer the “MAY-alias” question “YES”

Finally, let’s look at whether v.events[0] and v.events[2] MAY alias. Again, following the paths through the points-to graph, v.events points to the Object array node. And again, since we now have an array subscript to dereference, we take all arrows from this node labeled by an asterisk in parallel. So, according to this graph, v.events[0] could point to either the Floor node or the Event node. And v.events[2] could also point to either the Floor or Event node. Since the sets of nodes they could point to overlap, we cannot prove using this heap abstraction that v.events[0] and v.events[2] do not alias. So we answer “YES” to the question of whether they MAY alias.

SEGMENT

Modeling Aggregate Data Types

Modeling Aggregate Data Types: Arrays

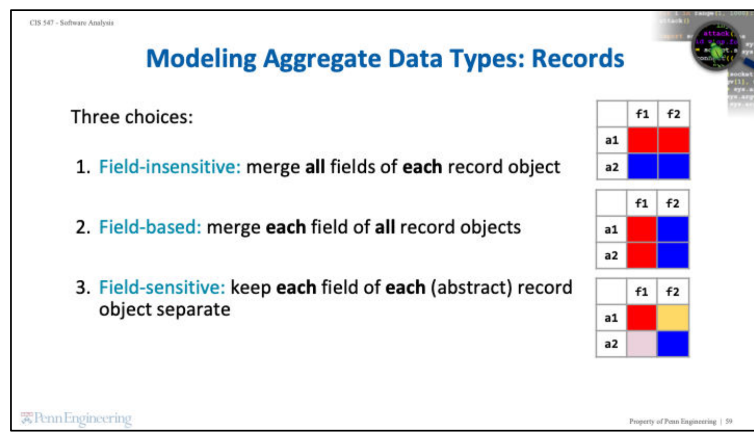
- Common choice: single field [*] to represent all array elements
  - Cannot distinguish different elements of same array
- More sophisticated representations that make such distinctions are employed by array dependence analyses
  - Used to parallelize sequential loops by parallelizing compilers

Another dimension for abstracting program data in a pointer analysis concerns how to model aggregate data types. These include arrays and records. Let's look at arrays first.

A common choice in pointer analyses is to use a single field, denoted by asterisk, to represent all elements of an array. Such analyses therefore cannot distinguish between different elements of an array. The pointer analysis algorithm we saw earlier in this lesson uses this representation.

More sophisticated representations make distinctions between different elements of an array. Such representations are employed in so-called array dependence analyses whose primary goal is to determine whether two integer expressions refer to the same element of an array, much like the primary goal of pointer analysis is to determine whether two pointer expressions alias.

Array dependence analysis is used in parallelizing compilers to parallelize sequential loops.



Now let's look at the another common kind of aggregate data type: records.

Records are also known as structs in C or classes in object-oriented languages like C++ and Java.

There are three common choices that pointer analyses use to model records.

We will illustrate these three choices using a record type with two fields f1 and f2. Suppose there are two objects a1 and a2 of this record type.

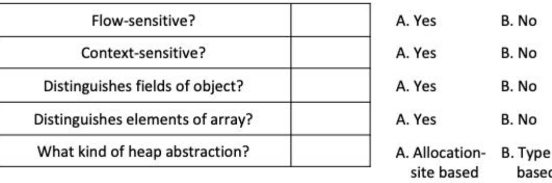
The first option shown here is a field-insensitive representation. The abstract state merges all fields of each record object. In other words, this representation prevents the analysis from distinguishing between different fields, such as f1 and f2, of the same record object, such as a1 or a2.

Another choice is to use a so-called field-based representation, which merges each field across all record objects. In other words, this representation prevents the analysis from distinguishing between the same field, such as f1 or f2, of different objects, such as a1 and a2.

The third choice is to use a field-sensitive representation, which keeps each field of each abstract record object separate. Assuming that a1 and a2 denote distinct abstract objects, this representation allows the analysis to distinguish all four memory locations denoted by this table: field f1 of a1, field f2 of a1, field f1 of a2, and field f2 of a2. It is easy to see that this is the most precise of the three representations. The pointer analysis algorithm we studied earlier in this module used this representation.

CIS 547 - Software Analysis

QUIZ: Pointer Analysis Classification



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

{QUIZ SLIDE}

Let's reflect on the pointer analysis we looked at. Classify it according to the dimensions we just discussed. In particular:

- Is the pointer analysis we discussed flow-sensitive?
- Is it context-sensitive?
- Did it distinguish between different fields of an object?
- Did it distinguish between different elements of an array?
- And what kind of heap abstraction did it use: allocation site-based or type-based?

Type in A to mean "Yes" and B to mean "No." For the last question, type "A" to mean "Allocation site-based" and "B" for "Type-based".

CIS 547 - Software Analysis

QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned:

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

{SOLUTION SLIDE}

Now let's review the answers.

Is the pointer analysis we discussed flow-sensitive? No, we used flow insensitivity as an approximation scheme to eliminate the complexity of tracking the points-to graph at each program point.

Is the pointer analysis context-sensitive? No. While the example program we used only had a single function, the algorithm would only analyze each function one time without regard to its context.

Did the pointer analysis distinguish different object fields? Yes, the algorithm did distinguish differently named object fields, such as the floors and events fields of the Elevator.

Did the pointer analysis distinguish different elements of an array? No, the algorithm abstracted away array elements using an asterisk to represent all subscripts.

And what kind of heap abstraction did the pointer analysis use: allocation site-based or type-based? Though we discussed type-based abstraction later in the lesson, the algorithm we discussed used an allocation site-based heap abstraction.

CIS 547 - Software Analysis

SEGMENT

Stack-Based Pointer Analysis

PennEngineering Property of Penn Engineering | 62

CIS 547 - Software Analysis

Stack-directed Pointers

- Two kinds of pointers: heap-directed and stack-directed
  - Heap-directed: p = new … or p = malloc(…)
  - Stack-directed: p = &v
- So far in this module: heap-based pointer analysis
- Next, we will define a stack-based pointer analysis

PennEngineering Property of Penn Engineering | 63

In general, a program's memory consists of static memory, called the stack, and dynamic memory, called the heap. Thus, pointers are of two kinds: heap-directed or stack-directed, depending upon the kind of memory that they point to. A heap-directed pointer p is created by using a construct such as the new keyword in Java or a call to malloc in C. A language like Java only provides support for heap-directed pointers. Thus, it is suffices for a pointer analysis for Java programs to reason about only heap-directed pointers.

On the other hand, for a language like C, a pointer analysis must also reason about stack-directed pointers. In C, a stack-directed pointer p is created by using the address-of operator, and it takes as its value the address on the stack of the referred variable v. This creates an entire class of pointers that we must also handle in order to reason about points-to relationships.

So far in this module, we focused on heap-directed pointers. Pointer analyses which reason about such pointers are called heap-based.

Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation sites. It is therefore important for pointer analyses of C programs to also analyze stack-directed pointers.

We will next look at pointer analyses that reason exclusively about stack-directed pointers. Such pointer analyses are called stack-based. The heap-insensitive

abstraction scheme we looked at is typically used by such pointer analyses in that they abstract all dynamic memory using a single abstract memory location.

Kinds of Statements

(statement) s ::= v1 = &v2
| v1 = v2
| v1 = *v2
| *v1 = v2

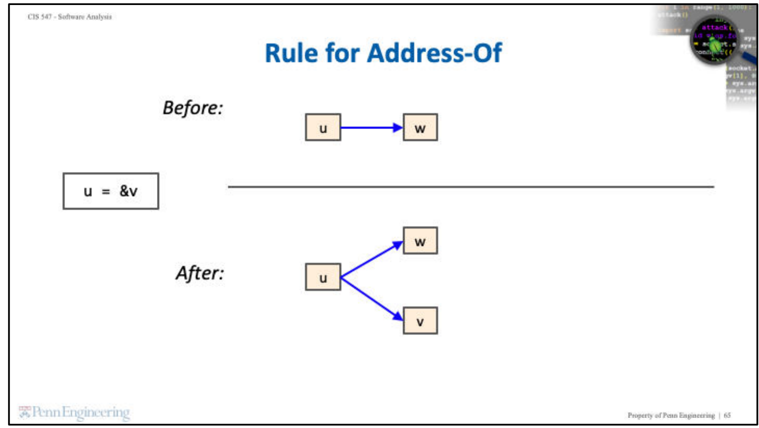
(variable) v

For a stack-based pointer analysis, it suffices to consider the following kinds of statements, which are analogous to those for heap-based pointer analysis.

First, we must consider the "Address-of" statement which creates a pointer using the & operator.

In addition to pointer creation, we also must consider when a variable aliases that pointer. This situation is shown by the assignment v = v2. We call this a copy statement because the memory location is copied from v2 to v1.

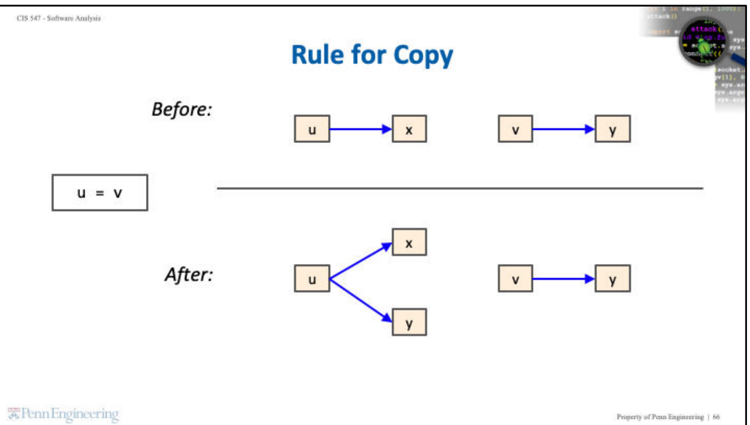
Next we must consider reads from and writes to memory locations denoted by pointer variables. We call these "indirect-reads" and "indirect writes." This is intuitive as we are accessing the memory locations indirectly through a pointer.



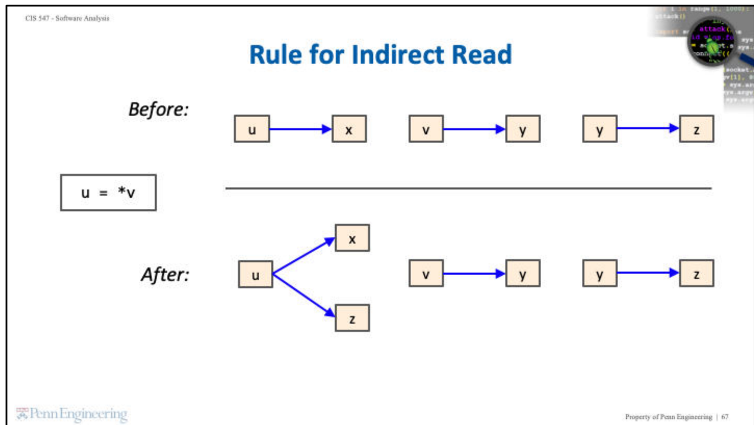
Let us now define the rules of Andersen's algorithm for stack-based pointer analysis similar to the rules we saw earlier in this module for heap-based pointer analysis.

Recall that the algorithm constructs a points-to graph by repeatedly updating it depending on the type of statement currently being analyzed. We will look at the rule for each of the four kinds of statements in turn.

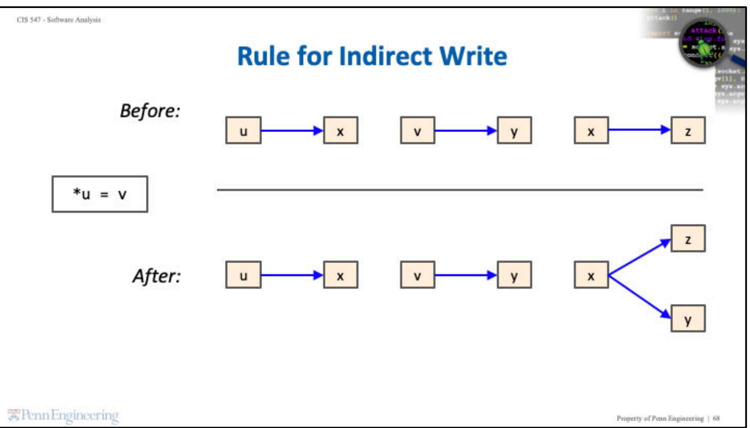
The rule for the address-of statement wherein the address of v is assigned to u states that u may point to v. Note that we are performing a weak update since our analysis is flow-insensitive: so any variables that u already points to, such as w, continue to be pointed to by u.



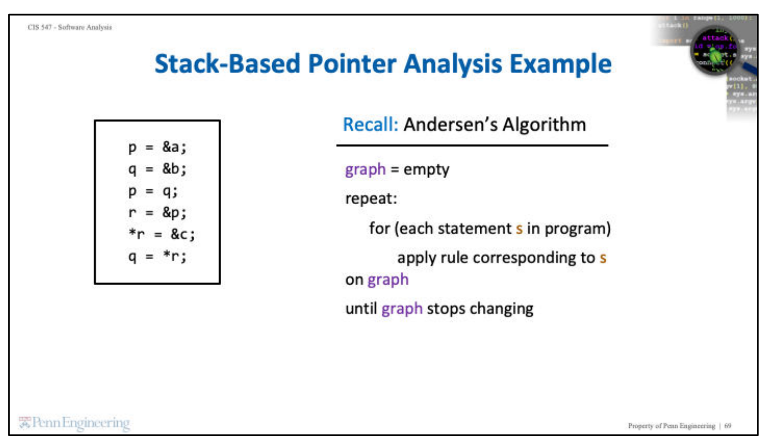
Let us next look at the rule for copy statements. When v is assigned to u, any variables pointed to by v, such as y, may also be pointed to by u. Notice that once again we perform a weak update: in particular, if u pointed to some variable x before, then u continues to point to x after the application of this rule.



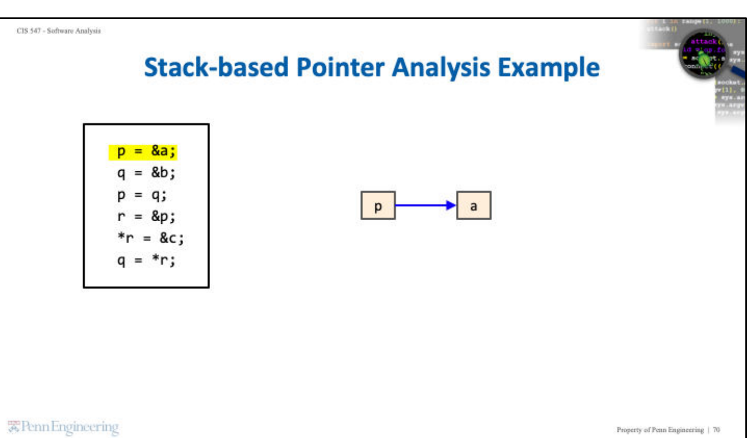
We next look at the rule for indirect reads. Suppose v may point to y and y may point to z. Then, after this indirect read, u may point to z. As before, we perform a weak update: if u pointed to some variable x before, then u continues to point to x after the application of this rule.



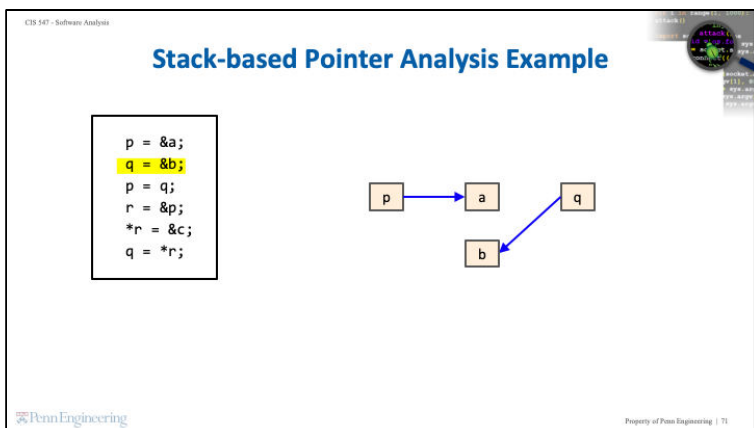
Lastly, let’s look at the rule for indirect writes. Suppose that u may point to x and v may point to y. Then, after this indirect write, x may point to y. As before, we perform a weak update: if x pointed to some variable z before, then x continues to point to z after the application of this rule.



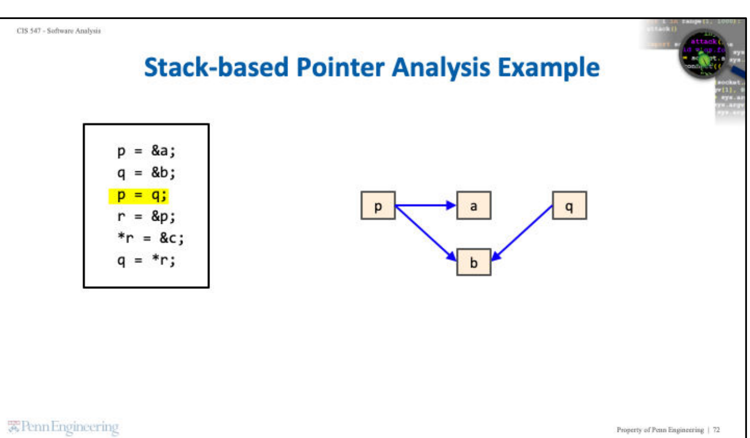
Let us apply our stack-based pointer analysis to an example C program. Recall from earlier in this module that Andersen's algorithm starts with an empty points-to graph and, in each iteration of the outer loop, applies the rule for each statement in the program, until the graph stops changing. Since our example program does not have any loops, a single iteration of the outer loop will suffice if we visit the statements in order of program execution.



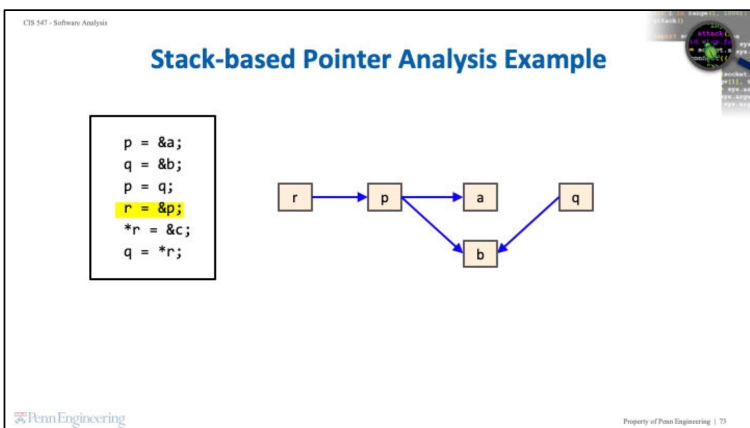
From the rule for the first statement in the program, an address-of statement, the graph is updated to capture the fact that p may point to a.



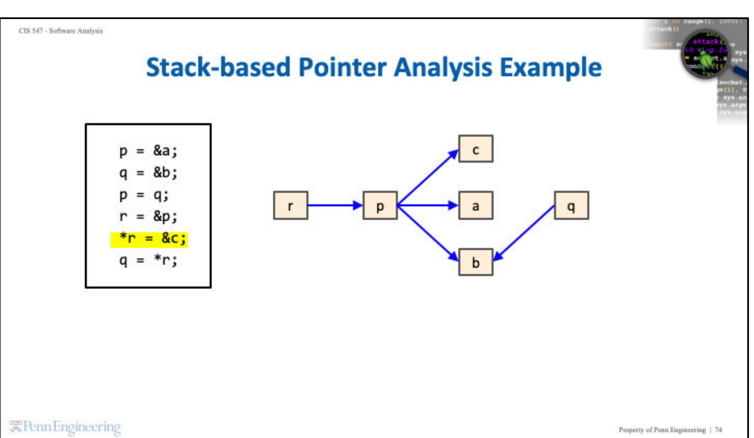
Applying the same rule to the second statement in the program, also an address of statement, the graph is updated to capture the fact that q may point to b.



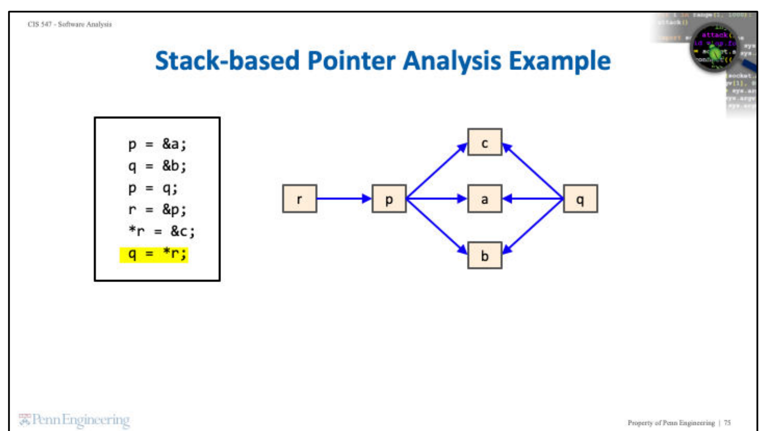
The third statement in the program is a copy statement. If q may point to some variable, then p may point to that variable. Since q points to b, we make p point to b as well.



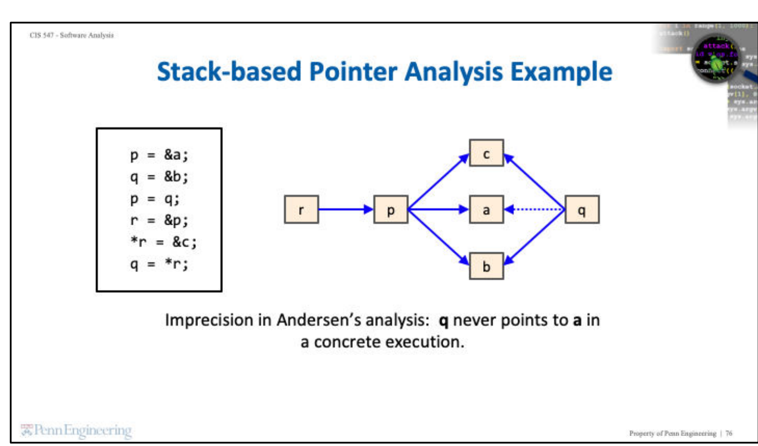
The next statement is again an address-of statement, and we update the graph to make r point to p.



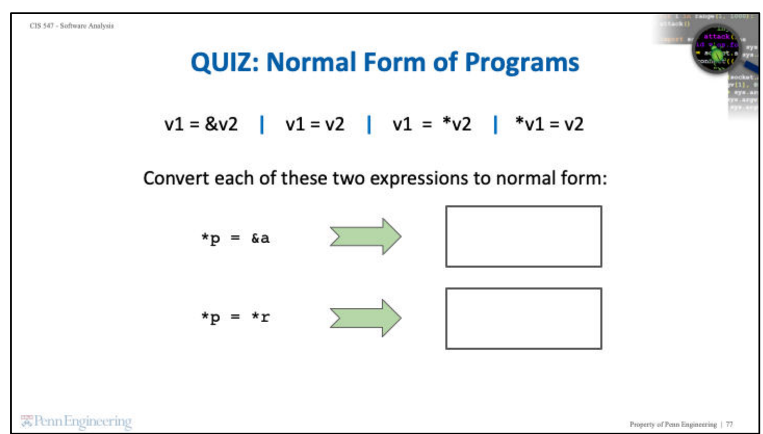
The next statement comprises an address-of statement followed by an indirect write statement. Applying the rules for both of those statements, we derive that p may point to c. Notice that, because it is an indirect write, the points-to information for r itself does not change: rather, the points-to information for variables pointed to by r, in this case just p, is affected.



The last statement in the program performs an indirect read. Since r may point to p, and p may point to a, b, and c, it follows from the rule for indirect reads that q may point to a, b, and c as well. Since the graph already captures the fact that q may point to a and b, we simply establish the points-to relationship from q to c. This is the final points-to graph constructed by Andersen's stack-based pointer analysis for our example program. Any further iterations will not result in additional updates to the points-to relationships.



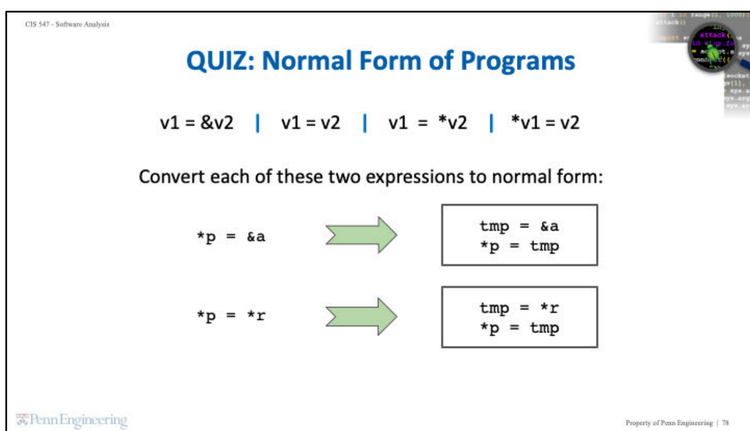
Andersen's analysis is sound but incomplete. An example of the kind of false positive it can produce is the points-to relationship "q arrow a". In a concrete execution of this C program, q can only point to b or c at any given program point, but never a.



{QUIZ SLIDE}

The four kinds of statements we introduced for stack-based pointer analysis are sufficient to represent all the operations that we might need to reason about in order to determine whether two pointers may alias.

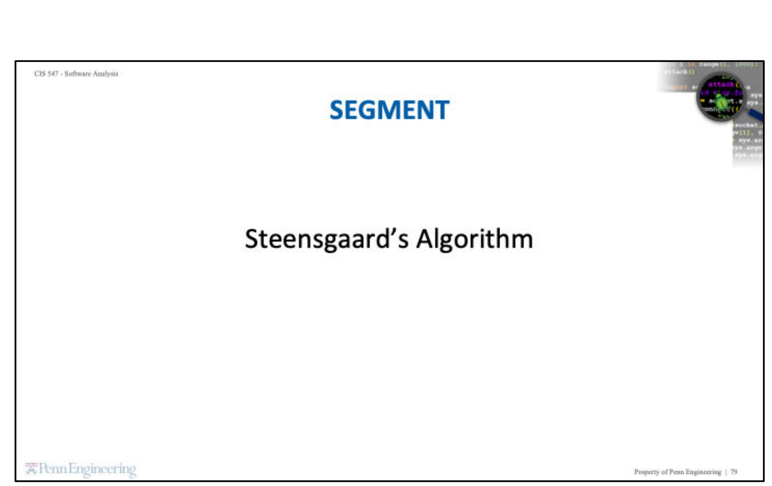
Show how each of the following two expressions can be replaced by equivalent programs using only the four kinds of statements above.

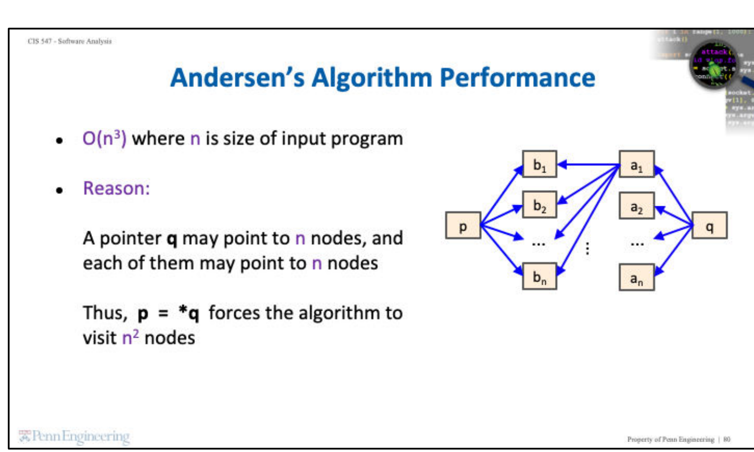


{SOLUTION SLIDE}

The first expression takes the address of variable a and stores it in the memory location denoted by pointer variable p. It can be decomposed into an equivalent sequence comprising the following two statements: the first takes the address of a and stores it in a variable named tmp, and the second takes the value of tmp and stores it in the memory location denoted by pointer variable p.

The second expression copies the value in the memory location denoted by pointer variable r to the memory location denoted by pointer variable p. It can be decomposed into an indirect read followed by an indirect write, with the copied value being transferred through a variable named tmp.





Andersen's algorithm is sufficiently precise in practice but this precision comes at the cost of performance. The worst-case time complexity of Andersen's algorithm is n^3 where n is the size of the input program. The following points-to graph elucidates the intuition for this. In this graph, a pointer q points to n nodes, a1 through an, and each of them points to n nodes; for instance, a1 points to b1 through bn. Then, the indirect read statement "p = * q" forces the algorithm to visit n^2 nodes. Since the program size is n, there can be upto n such statements, which yields a worst-case running time of n^3.

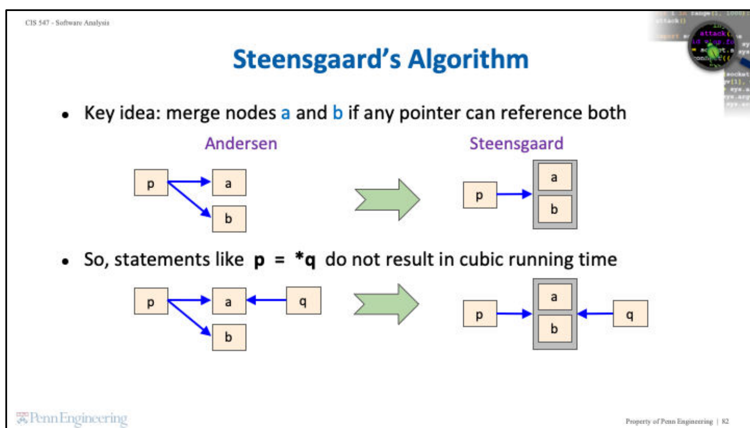
Scalable Pointer Analysis

- Andersen's algorithm is slow
  - Worst-case: $O(n^3)$ where n is size of program
  - In practice: $O(n^2)$
- Steensgaard's algorithm: a faster but less precise solution
  - Worst-case: $O(n . \alpha(n))$, or "almost linear" time
  - Implemented in LLVM Alias Analyzer

Although the example demonstrating the worst-case cubic time complexity of Andersen's algorithm was a pathological one, in practice, Andersen's algorithm runs in time quadratic in the size of the input program, which is still prohibitively expensive when analyzing very large programs.

A popular alternative in the literature is Steensgaard's algorithm. It trades precision for speed. The worst-case time complexity of Steensgaard's algorithm is almost linear in the size of the input program. Specifically, it is n times alpha n, where alpha is the inverse Ackermann function – an extremely slow-growing function that, for any practical input size n, is a small constant less than 5.

Steensgaard's algorithm is implemented in LLVM's alias analyzer and has been shown to analyze programs with millions of lines of code in under a minute.

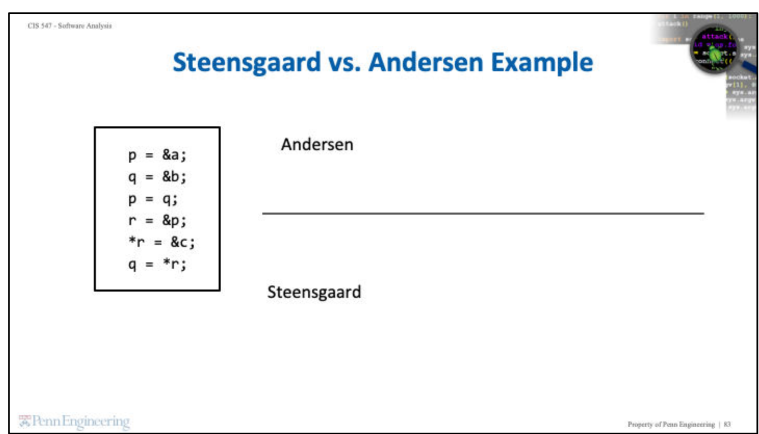


The key idea underlying Steensgaard's algorithm is to assign a synthetic type to each pointer. That is, each pointer references nodes of a specified type. If a pointer p may reference two nodes a and b, they must be of the same type. Thus, we merge them into a single node in the points-to-graph.

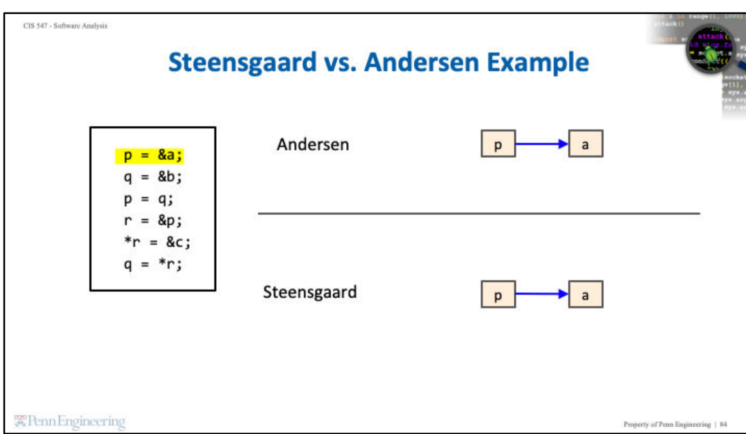
The algorithm can be viewed as performing type inference through a process called unification, whose worst case time complexity is almost linear. We will learn about type inference and unification in a future module.

For now, it suffices to understand that Steensgaard's algorithm improves efficiency by representing the points-to-graph in linear space. Now, the update rule for indirect reads no longer causes a quadratic blowup, which in turn improves the running time performance.

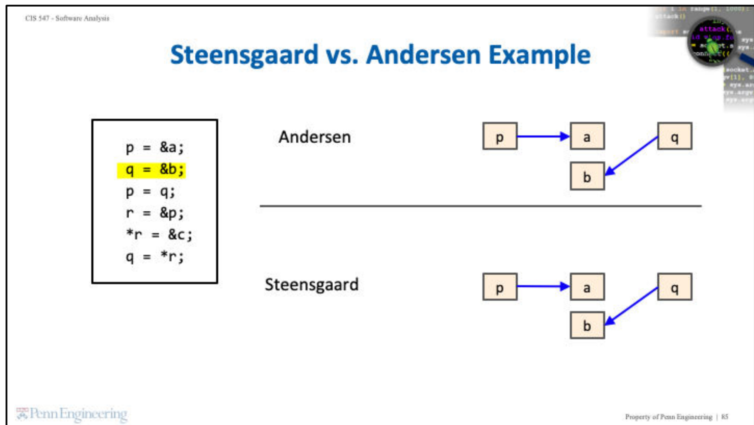
As usual in program analysis, this increase in efficiency comes at the expense of a decrease in precision. Consider the points-to-graph on the left, created by Andersen's algorithm, and the corresponding points-to graph on the right, created by Steensgaard's algorithm. Suppose we were to ask both analyses whether q may point to b. According to Steensgaard's algorithm, the answer is yes. However, according to Andersen's algorithm, we see that it is indeed not possible for q to point to b.



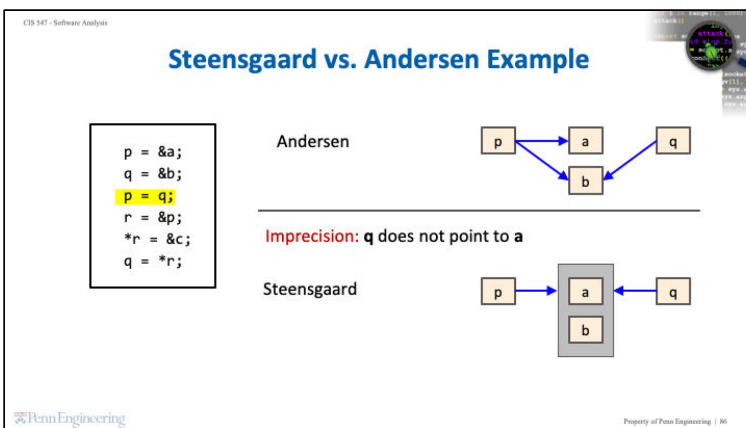
Let's revisit the example C program, this time simulating both Andersen's and Steensgaard's pointer analyses on it side-by-side.



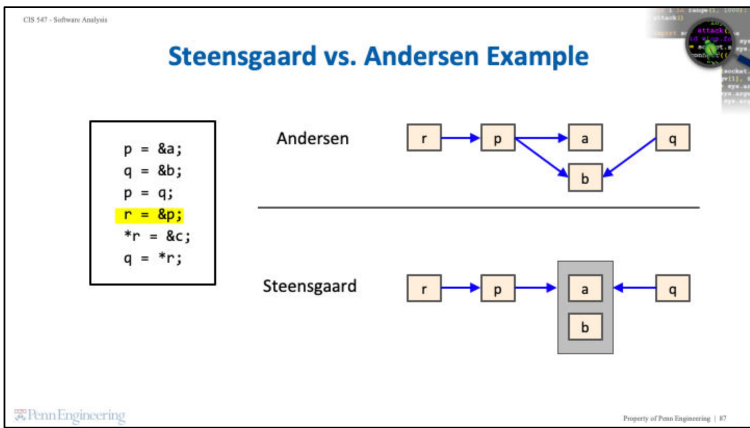
After analyzing the first statement, both the analyses update the points-to graph to capture the fact that p may point to a.



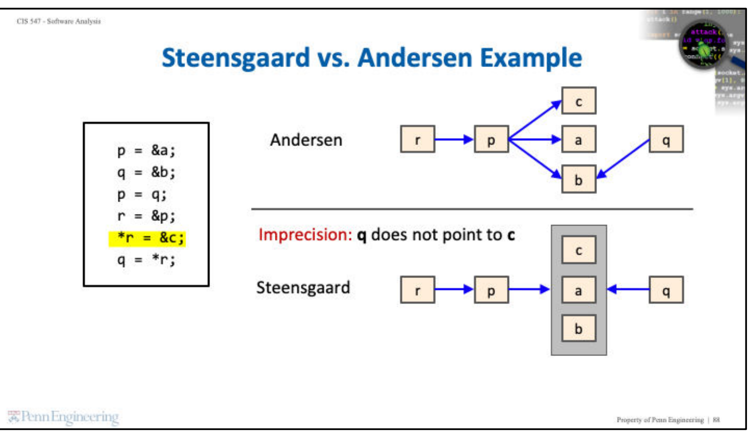
The second statement is similar, and both the analyses update the points-to graph to capture the fact that q may point to b.



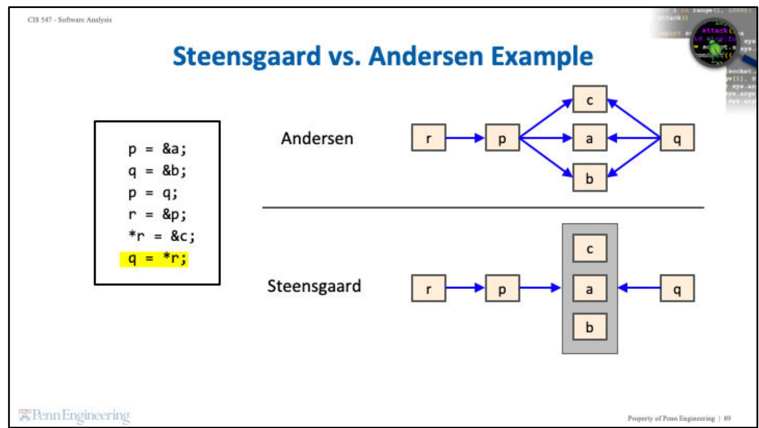
The third statement causes the analyses to update the points-to graph differently. Andersen's analysis captures the fact that p may point to a or b, but q may point to b but not a. However, Steensgaard's analysis merges a and b, which leads to imprecision: q is deemed to point to a or b, when in fact q cannot point to a.



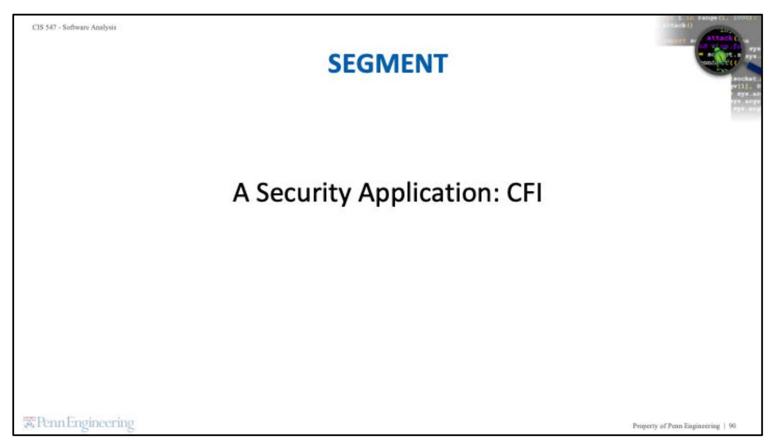
Continuing further, both analyses update their respective points-to graphs to capture the fact that r may point to p.



Next, the indirect write statement causes Andersen's analysis to capture the fact that p may point to c, but Steensgaard's analysis merges c with a and b, resulting in further imprecision: q is deemed to possibly point to c when in fact it cannot.



Lastly, the indirect read statement causes Andersen's analysis to capture the fact that q may point to whatever p may point to, which is a or b or c, while the points-to graph computed by Steensgaard's analysis remains unchanged. Although both the imprecisions that were encountered by Steensgaard's analysis relative to Andersen's analysis are now overcome, it is easy to construct an example where the result of Andersen's analysis is strictly more precise than that of Steensgaard's. For instance, this program without the last statement.

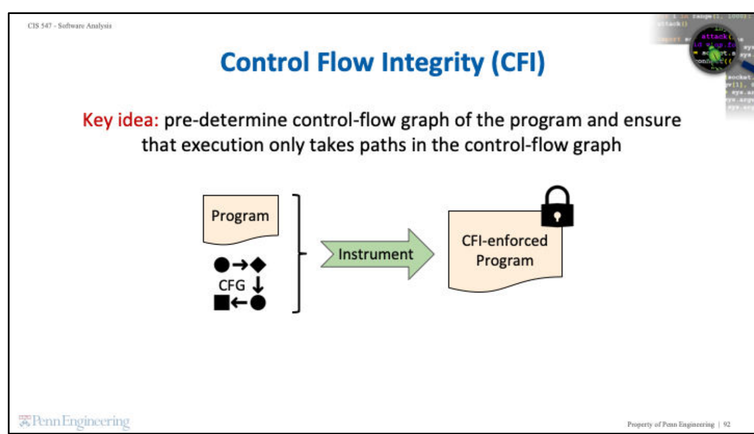


Security Application: Control Flow Integrity (CFI)

- A variety of malware attacks involve redirecting execution flow of program
- Solution: enforce a basic safety property called control-flow integrity, or CFI
  - More generally, a foundation for enforcing security policies
- Pointer analysis is needed to implement CFI effectively
- Many industrial-strength tools based on CFI, e.g. Microsoft's Control Flow Guard (built-in security mechanism in Windows 10)

We will now study an important and modern application of pointer analysis to software security called control-flow integrity. Current software attacks often build on exploits that hijack the control of the program execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily controlling program behavior. CFI enforcement is feasible in existing software applications via software rewriting. More generally, CFI provides a foundation for enforcing a variety of security policies.

Pointer analysis is needed to implement CFI efficiently as we shall see shortly. Many industrial-strength tools are based on CFI. An example is Microsoft's Control Flow Guard which is a built-in security mechanism in Windows 10.



The main idea for preventing hijacking the control of the program's execution is to to first define the expected program behavior and ensure the program does not deviate from it.

The CFI solution assumes that the expected behavior of the program is defined by its control-flow graph. Then, it rewrites the program with instrumentation to add runtime checks that ensure the execution is following the control-flow graph that was statically determined ahead of time.

Whenever an instruction transfers control, the destination must be valid according to the control-flow graph. In this manner, the program is secure even if the attacker has complete control over the address space.

CIS 547 - Software Analysis

Where Does Pointer Analysis Come In?

We need to resolve function pointers to build the control-flow graph

A reasonably precise pointer analysis computes that **d** points to **gt()**

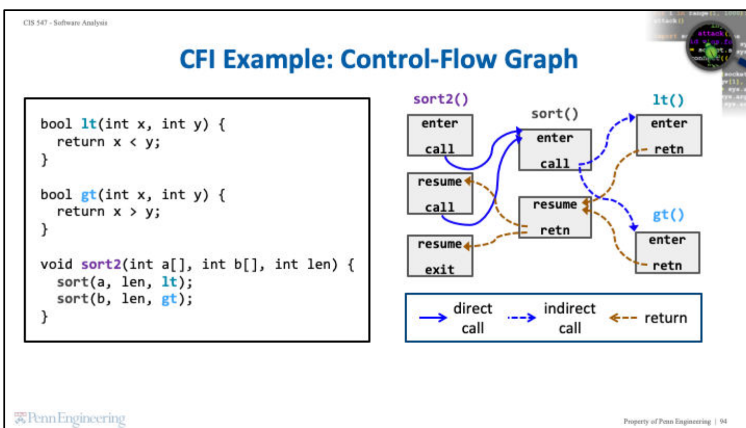
Either Andersen's or Steensgaard's algorithm can be used

```
void* a = (void*) &lt;
void* b = (void*) &gt;
void* c = (void*) &eq;
int **p = &a;
int **q = &b;
p = q;
int (*d)(int,int) = *q;
(*d)(...)    // ?
```

Penn Engineering

Property of Penn Engineering | 93

So where does pointer analysis come in? We need to resolve function pointers in order to statically build the control-flow graph. Consider the following example C program, which defines three function pointers a, b, and c, that point to functions called lt(), gt(), and eq(), respectively. Constructing the control-flow graph of this program requires determining which functions might possibly be called at runtime through the function pointer d. A reasonably precise pointer analysis would compute that d points only to function gt(). Either Andersen's or Steensgaard's algorithm can be used but note that, in this example, Steensgaard's analysis will conclude that c might point to gt() or lt(). While less precise than Andersen's result, it is still useful in that it can help rule out other functions such as eq() as possible targets.

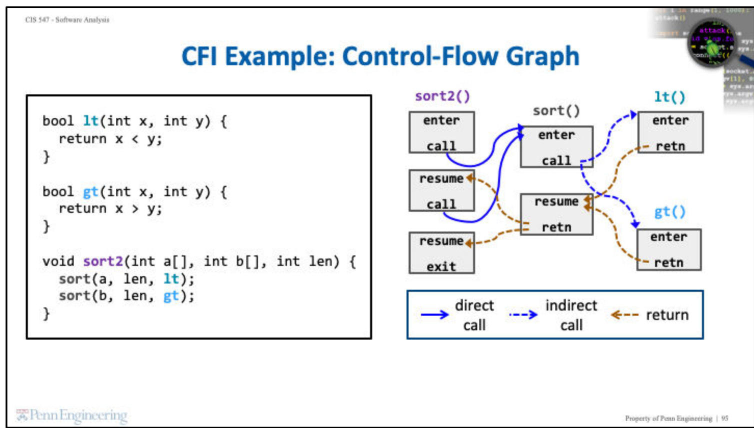


Let us understand CFI by means of an example. Here, we have a sort2() function, that sorts two arrays a and b, one ascending, and the other descending. It does so by calling another function called sort() that sorts a single array given a function pointer that defines the comparison for the sort operation. Here, the functions are lt() and gt(). The sort() function will invoke the function that is passed to it via a function pointer to perform the sorting.

On the right, we have the outline of the CFG that corresponds to the code fragment on the left. Solid blue lines are direct calls, dashed blue lines are indirect calls, and dashed red lines are returns. Let us trace through the flow of control in this program.

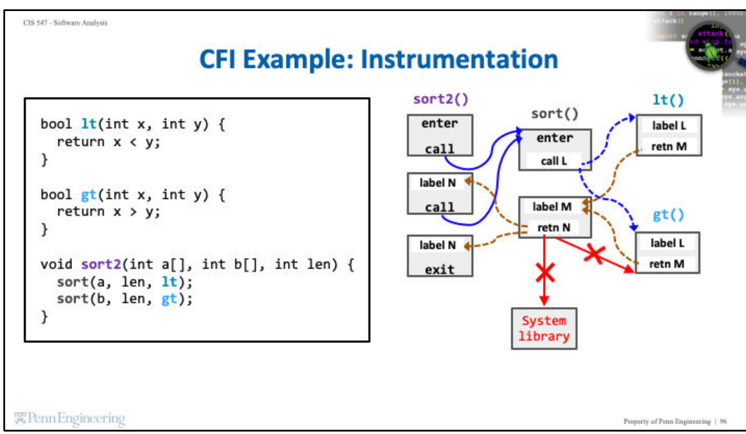
The sort2() function starts out by making a call to the sort function, which in turn calls the lt() function. The lt() function returns to the sort() function which finally returns to the sort2() function.

The sort2() function then makes another call to the sort function, which in turn calls the gt() function. The gt() function returns to the sort() function which finally returns to the sort2() function, and the flow is complete.



The goal of CFI is to ensure that, during runtime, the function calls and returns target addresses as described in the CFG. Therefore, we must estimate the call and return edges at compile time, and ensure that each one of these call and return jumps have the pre-determined destination.

Direct calls do not need to be monitored because they always have the same target, such as the two calls in the body of the sort2() function to the sort() function. However, the other edges in this CFG are indirect transfers due to function pointers. Note that return edges are always indirect transfers.



Next, we instrument the program to make sure the control transfer at runtime always adheres to the static control flow. We insert a label just before the target address of an indirect transfer. We also insert code to check the label of the target at each indirect transfer. If the label does not match, the program must be terminated.

Let's see an example of the labelling in this control-flow graph. The call targets for lt() and gt() must share the same label L. Likewise, the return targets for lt() and gt() must share the same label M, and the return targets for sort() must share the same label N.

So this prevents the control transfer to any dangerous spots such as system libraries. It also prevents the transfer from inside the program to an unintended spot in the same program. For instance, it prevents a return from sort() to gt().

CIS 547 - Software Analysis

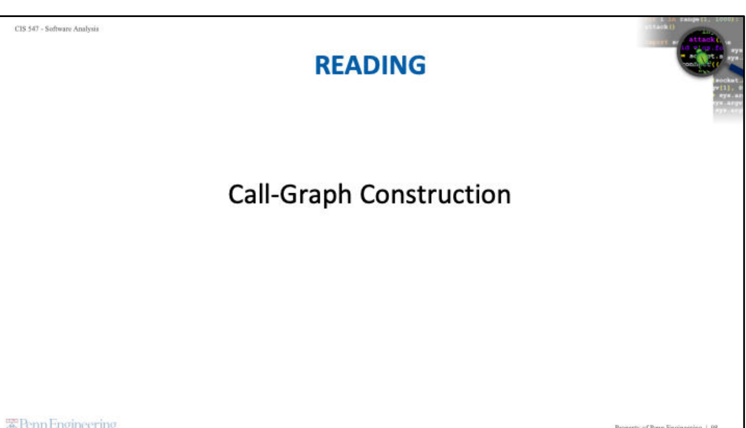
Where Did Pointer Analysis Help?

- Must instrument two kinds of indirect transfer controls …
    - Indirect transfers via **calls** through function pointers
    - Indirect transfers via **returns**
- … for which we must resolve function pointers in advance
    - Andersen's or Steensgaard pointer analysis algorithm

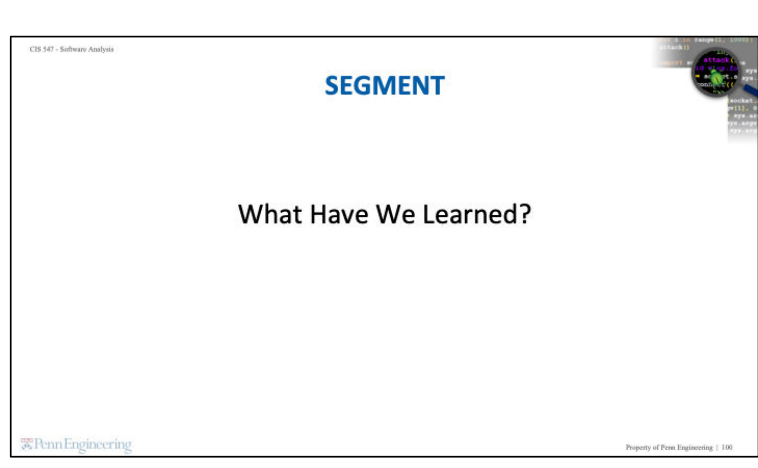
Penn Engineering

Property of Penn Engineering | 97

In summary, CFI requires instrumenting two kinds of indirect transfer controls: those via calls through function pointers, and those via returns. This requires resolving function pointers statically. Either Andersen's or Steensgaard's pointer analysis algorithm can compute the set of possible targets of function pointers.



CIS 547 - Software Analysis
LESSON
Review
Penn Engineering
Property of Penn Engineering | 99


CIS 547 - Software Analysis
SEGMENT
What Have We Learned?
Penn Engineering
Property of Penn Engineering | 100

CIS 547 - Software Analysis

What Have We Learned?

- What is pointer analysis?
- May-alias analysis vs. must-alias analysis
- Points-to graph and Andersen's algorithm
- Classifying pointer analyses: flow sensitivity, context sensitivity, heap abstraction, aggregate modeling
- Stack-based pointer analysis and Seensgaard's algorithm
- Security application: Control Flow Integrity (CFI)

Let's recap the main topics that we have covered in this module.

You have seen, at a high level, what pointer analysis is: a procedure for proving facts of the form "pointer X and pointer Y do not alias."

You saw the difference between a MAY-alias analysis and a MUST-alias analysis. Since MAY-alias analysis is less technically demanding and typically more useful in practice, it is what we refer to when we say "pointer analysis."

You saw what a points-to graph is and how it serves as an approximation of the actual state of data during a run of the program. This approximation allows us to conclude a pointer analysis in finite time at the cost of some number of false positives. You also used the points-to graph to decide whether two pointers MAY alias one another. You saw how the Andersen's algorithm works to build a points-to graph for a program.

You learned different dimensions along which pointer analysis algorithms may vary, including whether they remain sensitive to control flow and calling context, the kind of heap abstraction used, and how aggregate data is modeled (for example, in arrays).

You also learned about stack-based pointer analysis and how Steensgaard's algorithm creates a less precise points-to graph in a more efficient manner.

Lastly, we discussed an application of pointer analysis in security called control-flow integrity that can be used to enforce various security policies.

With this knowledge, you should be able to implement a pointer analysis algorithm for a simple programming language, and you should be able to determine the costs and benefits of different types of pointer analyses.

In this module, we presumed that the program we are analyzing consists of a single procedure. Realistic programs, on the other hand, consist of many procedures calling one another. Therefore, any realistic dataflow analysis must be able to reason about programs with multiple procedures. Many different approaches exist to extend a dataflow analysis from single-procedure programs to programs with multiple procedures. In the next module, we will learn about these approaches. Following the theme we have seen in this module and the previous one, all of these approaches will be sound but they will strike different tradeoffs between precision and efficiency; which approach to use thus depends on the desired tradeoff for the analysis you are designing.