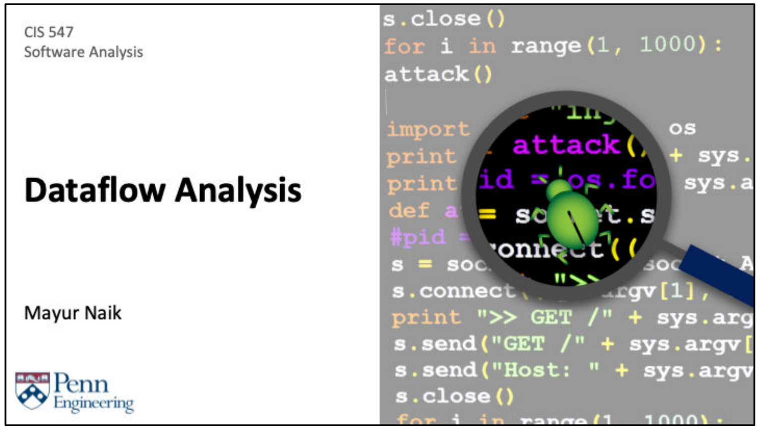


CIS 547
Software Analysis

Dataflow Analysis

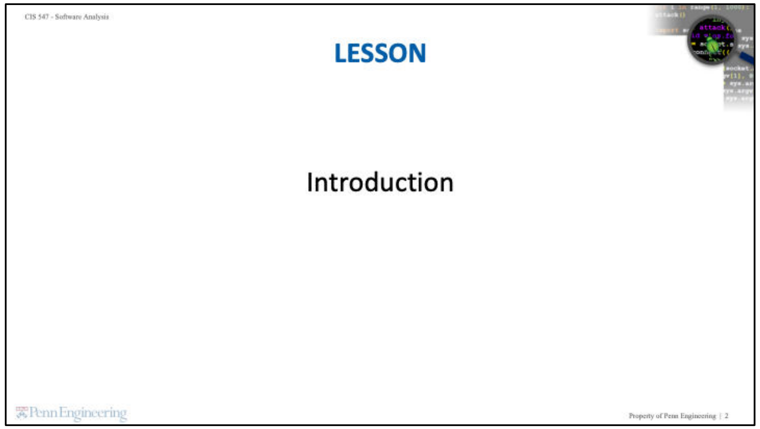
Mayur Naik

Penn Engineering

The field of software analysis is highly diverse: there are many approaches with different strengths and limitations in aspects such as soundness, completeness, applicability, and scalability.

In this module, we will introduce dataflow analysis, one of the dominant approaches to software analysis. We will see specific examples of useful dataflow analyses from the literature, and we will learn about a general technique to design a dataflow analysis.

After this module, you should be able to design your own dataflow analyses for a basic yet powerful programming language that captures the essence of realistic programming languages like C and Java.



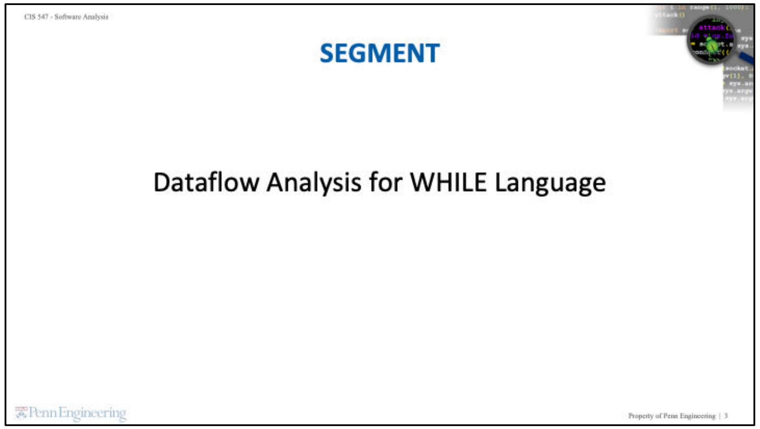
CIS 547 - Software Analysis

LESSON

Introduction

Penn Engineering

Property of Penn Engineering | 2



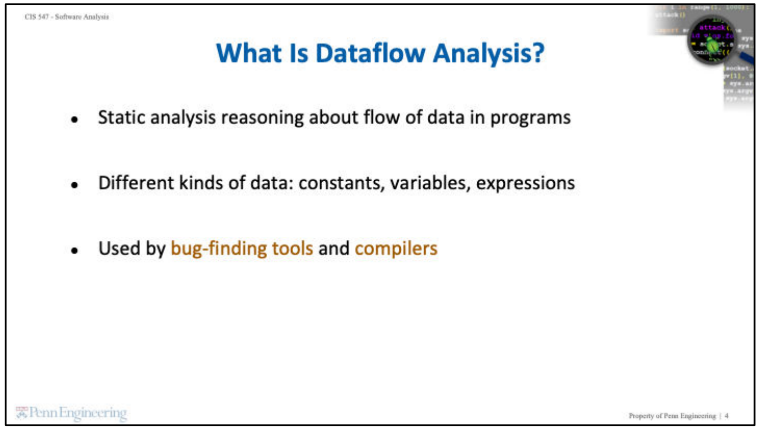
CIS 547 - Software Analysis

SEGMENT

Dataflow Analysis for WHILE Language

Penn Engineering

Property of Penn Engineering | 3



CIS 547 - Software Analysis

What Is Dataflow Analysis?

- Static analysis reasoning about flow of data in programs
- Different kinds of data: constants, variables, expressions
- Used by bug-finding tools and compilers

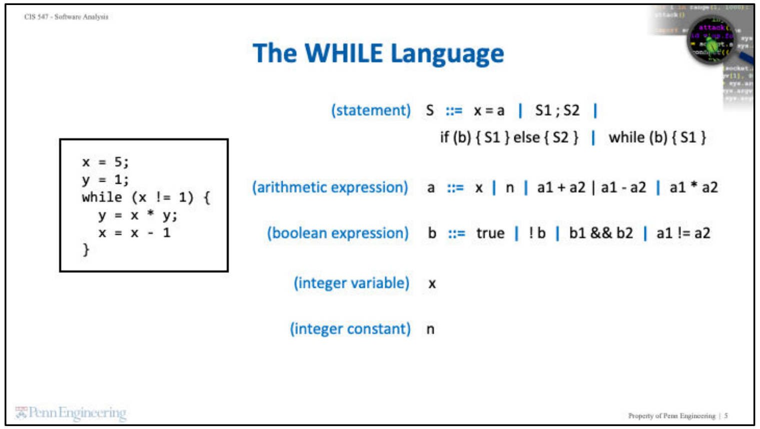
Penn Engineering

Property of Penn Engineering | 4

Dataflow analysis is a kind of static analysis for reasoning about the flow of data in runs of a program.

We can reason about data of different kinds. Some common examples are constants (such as the number 7 or the string literal “hello”), variables (such as ‘foo’), expressions (such as 7 * foo), and so on.

Dataflow analysis is used by bug-finding tools to find programming errors and is widely employed by compilers to generate optimized code.



The WHILE Language

(statement) S ::= x = a | S1 ; S2 |
if (b) { S1 } else { S2 } | while (b) { S1 }

(arithmetic expression) a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2

(boolean expression) b ::= true | ! b | b1 && b2 | a1 != a2

(integer variable) x

(integer constant) n

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Throughout this module, we will work with a simple programming language, called the WHILE language.

Here is an example program written in this language to compute the factorial of 5. The program has two integer variables x and y. It initializes these two variables and then updates them in a loop. Variable x contains the factorial of 5 at the end of the program.

Here is a formal grammar that precisely describes the syntax of programs written in the WHILE language.

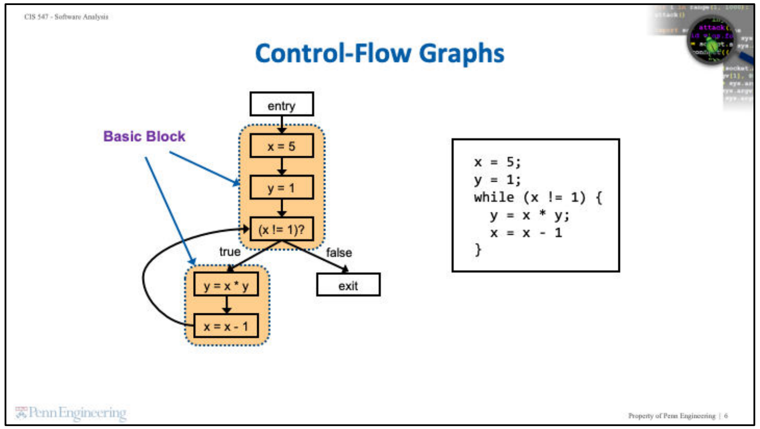
A program in this language is a statement S, which can be an assignment statement, a sequential composition of two statements, an if-then-else statement, or a while statement. Notice that this definition of a statement is recursive, so it can be used to describe arbitrarily large programs: programs with nested if-then-else statements, programs with nested loops, and so on.

For simplicity, we have only integer variables in this language. Furthermore, assignments to such variables can only be arithmetic expressions of a limited form. In particular, an arithmetic expression can be an integer variable which we denote using x, or an integer constant which we denote using n, or an addition of two expressions, or a subtraction of one expression from another expression, or a multiplication of two expressions.

The definition of arithmetic expressions is also recursive, allowing us to write programs with arbitrarily large expressions. It is easy to extend the syntax of these expressions to include other operators such as division, but we will leave those out for now to keep it simple.

Finally, to express conditions in if-then-else statements and while statements, we have boolean expressions. A boolean expression may be the constant true, the negation of another boolean expression, the conjunction of two boolean expressions b1 and b2, or a comparison between two arithmetic expressions a1 and a2. Again, to keep things simple, we allow limited kinds of boolean expressions, although it is easy to extend the syntax of this language to include operators besides the ones shown here.

Notice that the WHILE language does not have fancy constructs such as functions, pointers, or threads that are provided in commonly used programming languages like C and Java. This is because the presence of loops already makes the WHILE language expressive enough that interesting properties of programs written in this language are undecidable, and yet simple enough to allow us to study the fundamentals of dataflow analysis.



Control-Flow Graphs

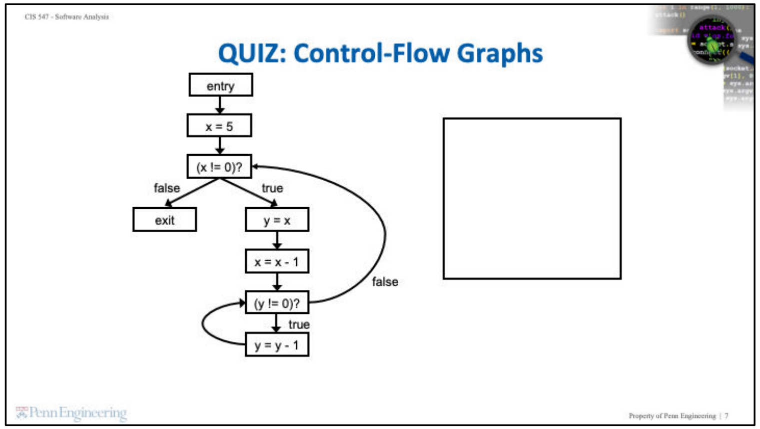
```
x = 5;
y = 1;
while (x != 1) {
   y = x * y;
   x = x - 1
}
```

Dataflow analysis typically operates on a suitable intermediate representation of the program. One such representation shown here, which we also saw earlier in the course, is a control-flow graph.

A control-flow graph is a graph that summarizes the flow of control in all possible runs of the program. Each node in the graph corresponds to a unique primitive statement in the program, such as an assignment or a condition test, and each edge outgoing from a node denotes a possible immediate successor of that statement in some run of the program.

Take a moment to convince yourself that this graph is indeed a control-flow graph of this program.

In practice, a maximal straight-line sequence of primitive statements, called a basic block, constitutes a single node. For instance, the nodes corresponding to the assignments x = 5, y = 1, and the test x != 1 form one basic block, and the nodes corresponding to the assignments y = x * y and x = x – 1 form another basic block. We will use these two representations interchangeably.

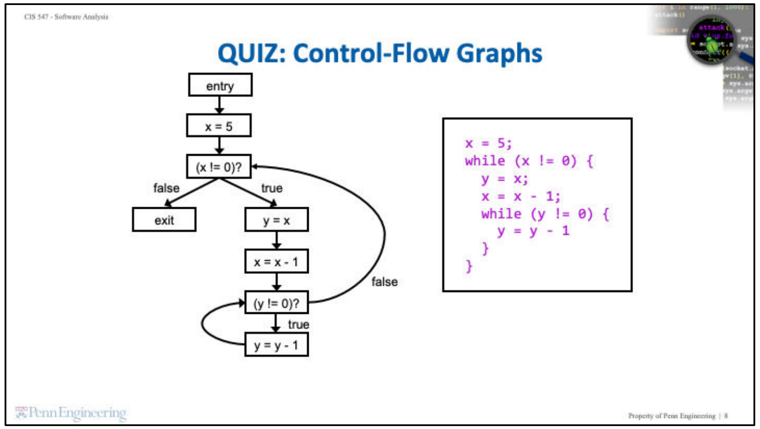


QUIZ: Control-Flow Graphs

{QUIZ SLIDE}

To check your understanding of control-flow graphs, let’s do an exercise converting a control-flow graph into the program it came from.

Here is a control-flow graph. In the adjoining box, write the program corresponding to this control-flow graph in the syntax of the WHILE language.



QUIZ: Control-Flow Graphs

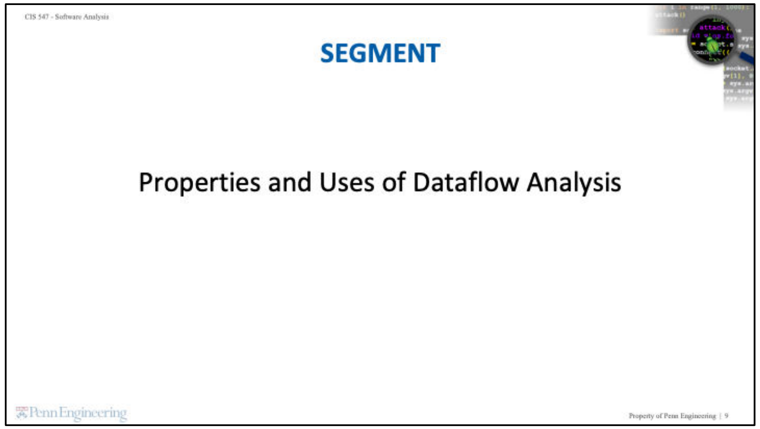
```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

{SOLUTION SLIDE}

The program corresponding to this control flow graph has two variables, x and y. It initializes the variable x to 5 and then executes a nested while statement.

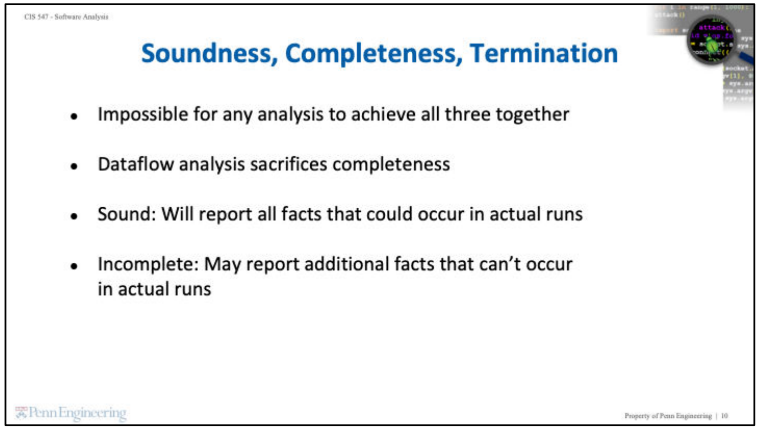
The outer while-loop decrements x in each iteration and terminates when x becomes 0. Also, at the start of each iteration, the variable y is initialized to the current value of x.

The inner while-loop decrements y in each iteration and terminates when y becomes 0.



SEGMENT

Properties and Uses of Dataflow Analysis



Soundness, Completeness, Termination

- Impossible for any analysis to achieve all three together
- Dataflow analysis sacrifices completeness
- Sound: Will report all facts that could occur in actual runs
- Incomplete: May report additional facts that can't occur in actual runs

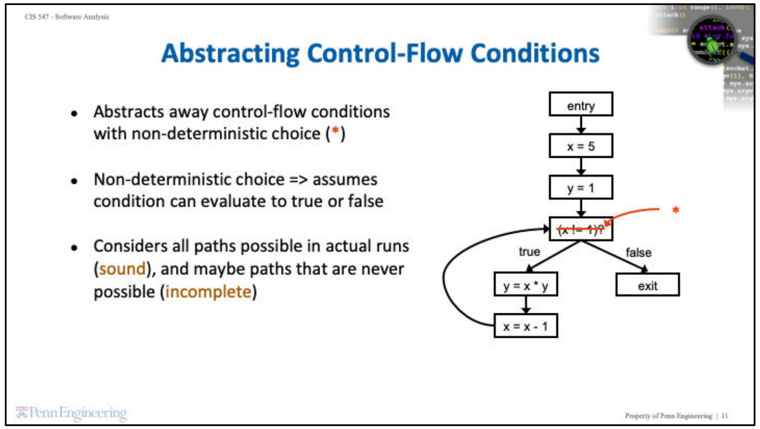
Recall that it is impossible to design a software analysis that is sound, complete, and guaranteed to terminate. This impossibility holds for dataflow analyses, as they are a kind of software analysis.

Dataflow analyses choose to sacrifice completeness to guarantee termination and soundness.

Since dataflow analysis is sound, it will report all dataflow facts that could occur in actual runs.

However, because dataflow analysis is incomplete, it may report dataflow facts that can never occur in actual runs.

Let's see next how a dataflow analysis achieves soundness by sacrificing completeness.



Abstracting Control-Flow Conditions

- Abstracts away control-flow conditions with non-deterministic choice (*)
- Non-deterministic choice => assumes condition can evaluate to true or false
- Considers all paths possible in actual runs (sound), and maybe paths that are never possible (incomplete)

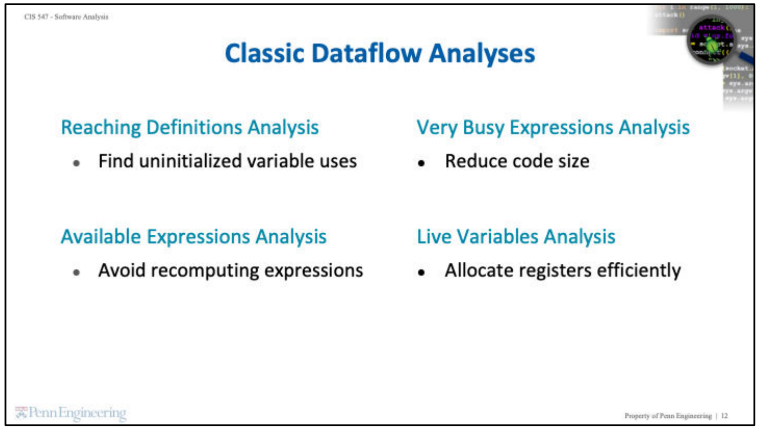
The primary source of incompleteness in dataflow analyses arises from abstracting away control-flow conditions with non-deterministic choice, which we will denote using the star symbol.

For this example program, dataflow analysis replaces the condition (x != 1) with non-deterministic choice. [Strike out boolean expression (x != 1) and replace it with a *.]

Non-deterministic choice simply means that the analysis will assume that the condition can evaluate to true or false, even if, for example, in actual runs the condition always evaluates to true.

By doing this, not only does a dataflow analysis ensure that it will consider all paths that are possible in actual runs of the program, and thereby guarantees soundness, but it also considers paths that are never possible in actual runs, which leads to incompleteness.

Next, let's look at some popular dataflow analyses and how they are used in compilers and bug-finding tools.



CIS 547 - Software Analysis

Classic Dataflow Analyses

Reaching Definitions Analysis

- Find uninitialized variable uses

Very Busy Expressions Analysis

- Reduce code size

Available Expressions Analysis

- Avoid recomputing expressions

Live Variables Analysis

- Allocate registers efficiently

Penn Engineering

Property of Penn Engineering | 12

We will illustrate how dataflow analysis computes properties on a control-flow graph through a series of four classic dataflow analyses in the literature. These analyses are: Reaching Definitions Analysis, Very Busy Expressions Analysis, Available Expressions Analysis, and Live Variables Analysis.

Before we dive into the details of these four analyses, let's take a look at four practical applications that motivate them.

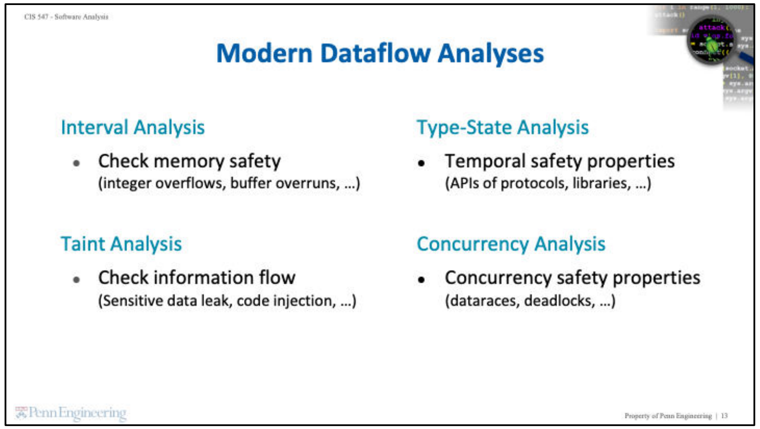
Reaching Definitions Analysis produces information that can be used by a bug-finding tool for discovering uses of potentially uninitialized variables in a program.

Very Busy Expressions Analysis computes information that can help reduce code size. This application can be critical to certain embedded devices that have code size constraints, such as pacemakers.

Available Expressions Analysis produces information that can be used by a compiler to avoid recomputing the same program expression multiple times in an execution, thereby producing code that runs faster.

Finally, Live Variables Analysis computes information that can be used by a compiler to efficiently allocate registers to program variables. Register allocation is the component of a compiler that most impacts the performance of the generated code.

We will dive into the details of each of these four analyses in this module.



CIS 547 - Software Analysis

Modern Dataflow Analyses

Interval Analysis

- Check memory safety
  (integer overflows, buffer overruns, …)

Type-State Analysis

- Temporal safety properties
  (APIs of protocols, libraries, …)

Taint Analysis

- Check information flow
  (Sensitive data leak, code injection, …)

Concurrency Analysis

- Concurrency safety properties
  (dataraces, deadlocks, …)

Penn Engineering

Property of Penn Engineering | 13

Besides the classic dataflow analyses that one might expect in the literature on compilers, dataflow analyses also underlie a variety of tools that are used in the software industry today for improving software quality. Some of these modern dataflow analyses include Interval Analysis, Type-State Analysis, Taint Analysis, and Concurrency Analysis.

Interval Analysis is used to check various memory safety properties of programs. It involves reasoning about the ranges of integer variables and can therefore check for integer overflows, buffer overruns, and other errors involving integer-related data. A popular tool that employs interval analysis is the ASTREE static analyzer for C programs; it is routinely used by Airbus, BMW, and other companies that develop or use embedded software. We will delve into interval analysis towards the end of this module.

Type-State Analysis is used to check for temporal safety properties that we discussed in the module on software specifications. Recall that such properties can be used to specify correct usage of APIs of protocols and libraries, for example, that a lock must be acquired and released in strict alternation. A popular tool that employs type-state analysis is the SAFE tool by IBM for enterprise Java applications; we will learn more about it in a subsequent module on pointer analysis.

Taint Analysis is used to check for information-flow properties. These properties concern tracking the flow of data through programs from untrusted sources, such as inputs that an attacker can control, to sensitive sinks, whose compromise can lead to leaking sensitive data or injecting malicious code. Examples of tools that employ taint analysis include those used to vet newly uploaded or updated mobile applications on marketplaces for Android and iOS devices. Mobile applications are a prime target for advertently or inadvertently misusing sensitive information of their users.

Finally, Concurrency Analysis is used to check for concurrency safety properties such as dataraces and deadlocks. As we saw in the Cuzz tool in the module on Random Testing, concurrency bugs are notoriously hard to uncover using testing, because they typically manifest under highly specific schedules that are dictated by the underlying thread scheduler. An example of a tool that uses dataflow analysis to find concurrency bugs is Facebook's RacerD for finding dataraces is concurrent Java programs.

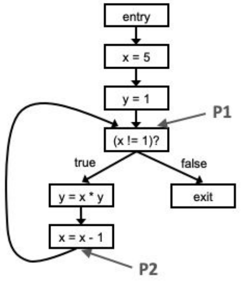
SEGMENT

What Is Reaching Definitions Analysis

Reaching Definitions Analysis

Goal: Determine, for each program point, which assignments have been made and not overwritten, when execution reaches that point along some path

- "Assignment" == "Definition"



We will use Reaching Definitions Analysis to introduce the key concepts of dataflow analysis.

Each dataflow analysis has a goal that specifies the kind of data flow information that the analysis computes. The goal of reaching definitions analysis is to determine which assignments might reach each program point. More accurately, this analysis determines, for each program point, which assignments potentially have been made and not overwritten, when the program's execution reaches that point along some path.

For the purpose of this analysis, we will use the terms "assignment" and "definition" interchangeably, since an assignment corresponds to a definition in the WHILE language.

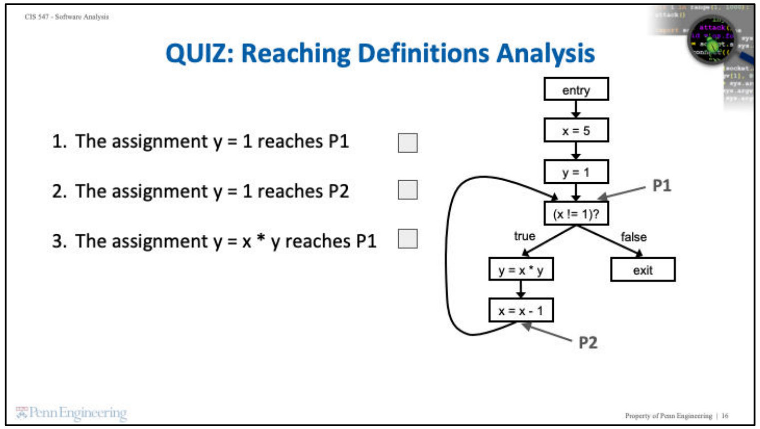
Let us look at the following example program. There are four definitions in this program: x = 5, y = 1, y = x * y, and x = x - 1.

Consider two program points: P1, at the entry of this condition, and P2, at the exit of this assignment.

Let's consider the definition x = 5. This definition reaches point P1 as there is no overwriting assignment to x along this path.

But this definition does not reach point P2, as x is overwritten by assignment x = x - 1 every time execution reaches P2.

Take a moment to understand the goal of reaching definitions analysis. We will next do a quiz to practice a few more reaching definitions in this control-flow graph.



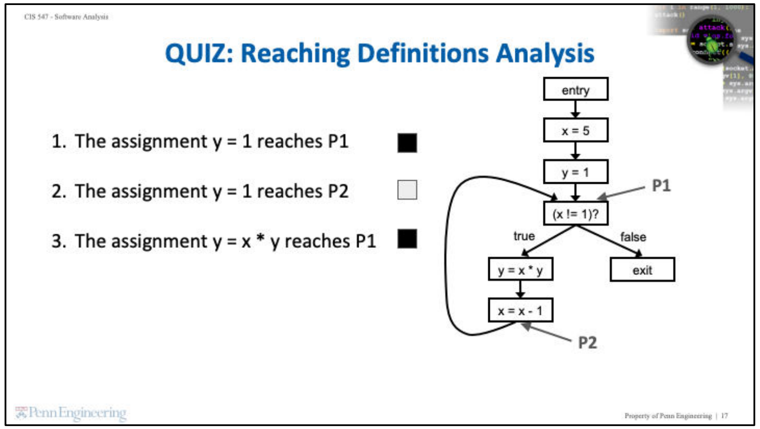
QUIZ: Reaching Definitions Analysis

1. The assignment y = 1 reaches P1 □
2. The assignment y = 1 reaches P2 □
3. The assignment y = x * y reaches P1 □

{QUIZ SLIDE}

To check your understanding of reaching definitions analysis, here’s a short quiz.

Check the boxes corresponding to the statements that are true about reaching definitions analysis.



CIS 547 - Software Analysis

QUIZ: Reaching Definitions Analysis

1. The assignment y = 1 reaches P1 ■
2. The assignment y = 1 reaches P2 □
3. The assignment y = x * y reaches P1 ■

Penn Engineering

Property of Penn Engineering | 17

{SOLUTION SLIDE}

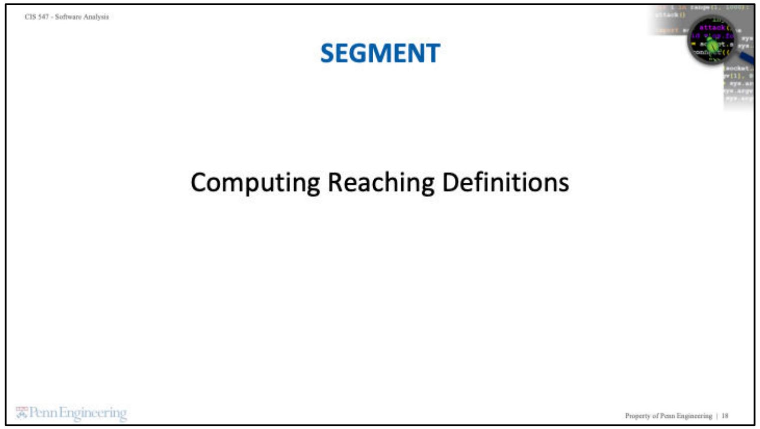
The 1st statement is True. Indeed, the definition y = 1 reaches P1 along this path. (gesture)

The 2nd statement is False. This is because y is overwritten by the definition y = x * y every time execution reaches P2. (gesture)

The 3rd statement is True. Indeed, the definition y = x * y reaches P1 along this path. (gesture)

Notice that different definitions can reach a given program point, as in the case of definitions y = 1 and y = x * y reaching P1. And conversely, a given definition can reach multiple program points, as in the case of y = x * y reaching both P1 and P2.

Next, let's dive into the details of the algorithm for reaching definitions analysis.



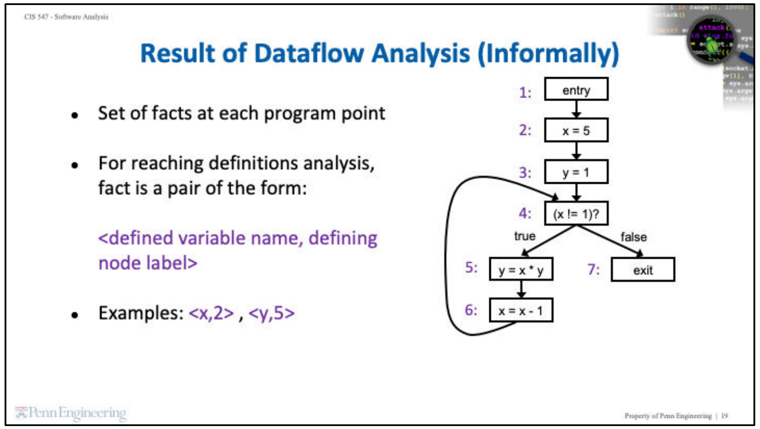
CIS 547 - Software Analysis

SEGMENT

Computing Reaching Definitions

Penn Engineering

Property of Penn Engineering | 18



CIS 547 - Software Analysis

Result of Dataflow Analysis (Informally)

- Set of facts at each program point
- For reaching definitions analysis, fact is a pair of the form:

  <defined variable name, defining node label>
- Examples: <x,2> , <y,5>

1:
entry
2:
x = 5
3:
y = 1
4:
(x != 1)?
true
false
5:
y = x * y
7:
exit
6:
x = x - 1

Penn Engineering

Property of Penn Engineering | 19

Informally speaking, the result of a dataflow analysis is a set of facts at each program point.

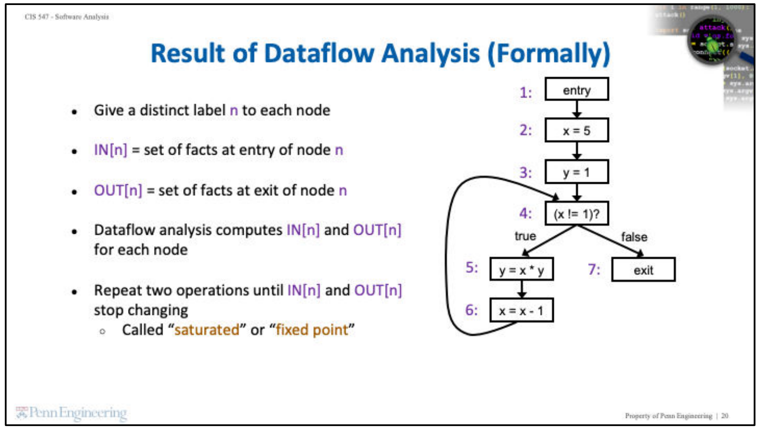
For example, reaching definitions analysis computes the set of definitions that may reach each program point.

To identify each program point uniquely, let's assign a distinct label to each node in the control-flow graph. Here, I've labelled the nodes in this control-flow graph from 1 through 7.

Then, we can denote each reaching definition as a pair comprising the name of the defined variable, along with the label of the node that defines it.

For example, the definition of variable x at the node labeled 2 is denoted as follows: <x, 2>. And the definition of variable y at the node labeled 5 is denoted as follows: <y, 5>.

Now, let's make this notions of a dataflow analysis result more precise.



CIS 547 - Software Analysis

Result of Dataflow Analysis (Formally)

- Give a distinct label n to each node
- IN[n] = set of facts at entry of node n
- OUT[n] = set of facts at exit of node n
- Dataflow analysis computes IN[n] and OUT[n] for each node
- Repeat two operations until IN[n] and OUT[n] stop changing
  - Called "saturated" or "fixed point"

1:
entry
2:
x = 5
3:
y = 1
4:
(x != 1)?
true
false
5:
y = x * y
7:
exit
6:
x = x - 1

Penn Engineering

Property of Penn Engineering | 20

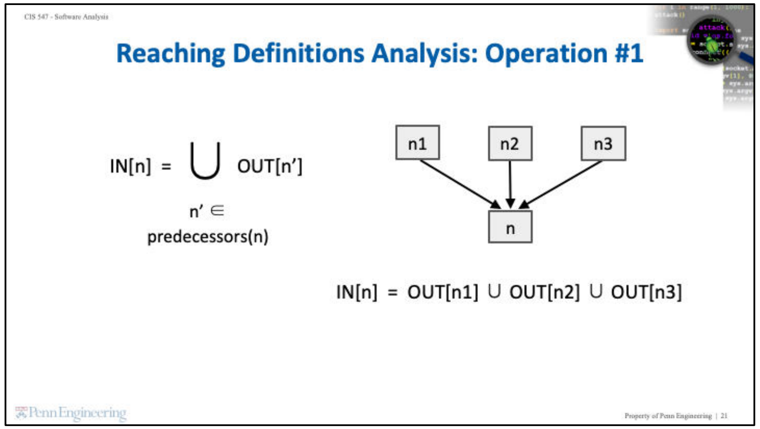
First, we give a distinct label n to each node.

For each node with label n, we use IN(n) to denote the set of facts at the entry of the node, and OUT(n) to denote the set of facts at the exit of the node.

A dataflow analysis computes the IN and OUT sets of facts for each node in the control-flow graph.

It does so by repeatedly applying two operations until the sets of IN and OUT facts for each node in the graph stop changing. At that point, we say that the result of the dataflow analysis is saturated or that it has reached a fixed point.

We will next introduce these two operations for reaching definitions analysis. Subsequently, we will see slight variations of these operations for the other three classic dataflow analyses.



Reaching Definitions Analysis: Operation #1

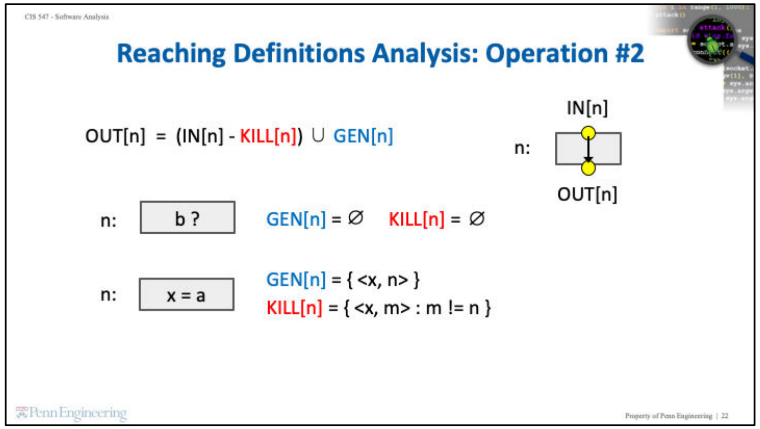
$$IN[n] = \bigcup_{n' \in \text{predecessors}(n)} OUT[n']$$

$$IN[n] = OUT[n1] \cup OUT[n2] \cup OUT[n3]$$

Here's the first of the two operations of reaching definitions analysis.

This operation states how to compute the set of facts at the entry of a particular node in the control-flow graph. We do this by taking the union of the sets of facts at the exit of that node's immediate predecessors. that is, the union of OUT[n'] for each immediate predecessor node n' of n.

For instance, suppose node n has three immediate predecessors n1, n2, and n3. Then, the set of facts at the entry of n is the union of the set of facts at the exit of n1, n2, and n3.



Reaching Definitions Analysis: Operation #2

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

n: b ? GEN[n] = ∅ KILL[n] = ∅

n: x = a GEN[n] = { <x, n> }
KILL[n] = { <x, m> : m != n }

The second operation of reaching definitions analysis tells us how to compute the set of facts at the exit of a particular node from the set of facts at the entry of that node. Unlike the previous operation that we just saw, this operation depends on the statement that occurs at the node we are looking at.

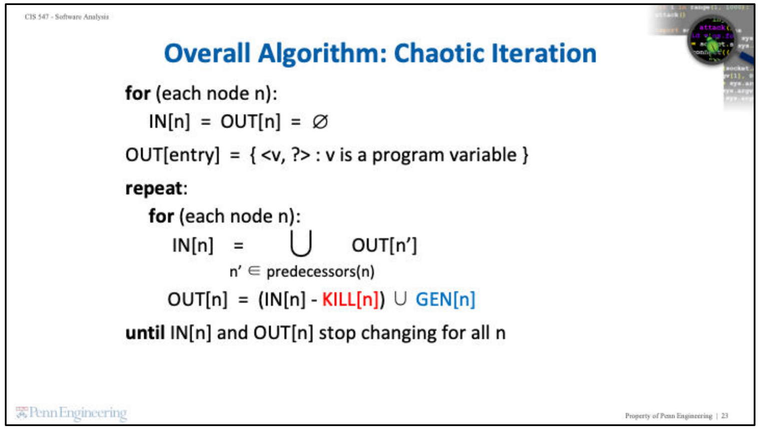
We'll first state this operation in its general form, and then we'll show specific instances of it corresponding to each kind of primitive statement in the WHILE language.

The general form of this operation states that the set of facts at the exit of node n is equal to the set of facts at the entry of node n, minus any definitions that are overwritten by node n, unioned with any new definitions that are generated by node n. We call these sets of definitions as the KILL set and the GEN set, respectively.

Determining the GEN and KILL sets requires knowledge of the statement that occurs at node n. For control-flow graphs of programs in the WHILE language, this statement can be either a condition or an assignment. Let's consider each of these two cases in turn.

If the statement at node n is a condition, then both the GEN and KILL sets are empty, since there are no definitions overwritten or generated by a conditional statement. (So, in this case, the set OUT[n] will equal the set IN[n].)

If the statement at node n is an assignment of the form x = a, then the GEN and KILL sets are more interesting. The GEN set contains the definition of variable x at node n itself, reflecting the fact that this definition is generated by node n. The KILL set, on the other hand, contains every definition of variable x except the one at node n, reflecting the fact that all those definitions will be overwritten by the one at node n. We denote this set compactly using set comprehension notation. You can review this notation by following the link: https://en.wikipedia.org/wiki/Set-builder_notation



Overall Algorithm: Chaotic Iteration

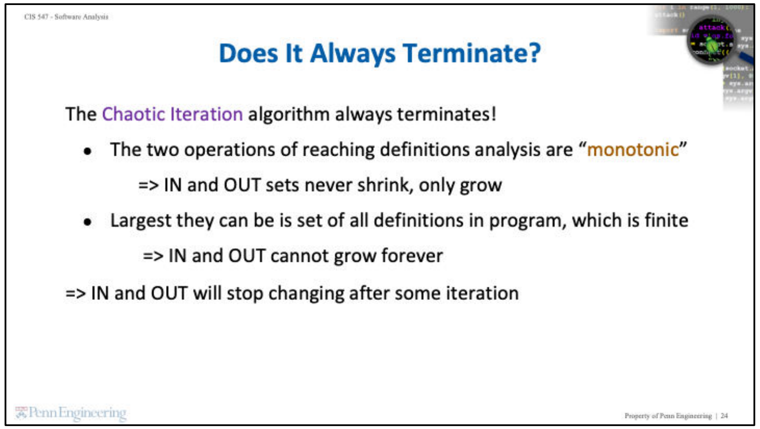
```
for (each node n):
    $IN[n] = OUT[n] = \varnothing$
$OUT[entry] = \{ \langle v, ? \rangle : v \text{ is a program variable} \}$
repeat:
    for (each node n):
        $IN[n] = \bigcup_{n' \in predecessors(n)} OUT[n']$
        $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$
until IN[n] and OUT[n] stop changing for all n
```

Equipped with the two operations of reaching definitions analysis, let’s step through the overall reaching definitions analysis algorithm.

The algorithm starts by initializing the IN and OUT set of each node n to the empty set. The only exception is the OUT set of the entry node, which is initialized to contain a hypothetical definition for each variable v in the program. It captures the fact that each variable is undefined, or uninitialized, at the start of the program.

The algorithm then performs its main task from which it derives the name chaotic iteration algorithm. The name highlights two important properties of the algorithm.

First, it is iterative: it repeatedly updates the IN and OUT sets of each node in the control-flow graph until they stop changing. Second, it is chaotic in the sense that in each iteration, it visits all nodes in the control-flow graph and applies the two operations we just discussed to update the IN and OUT sets of each node. Crucially, the order in which the nodes are visited does not matter, lending the adjective chaotic in the name of the algorithm.



Does It Always Terminate?

The Chaotic Iteration algorithm always terminates!

- The two operations of reaching definitions analysis are "monotonic"

  => IN and OUT sets never shrink, only grow
- Largest they can be is set of all definitions in program, which is finite

  => IN and OUT cannot grow forever

=> IN and OUT will stop changing after some iteration

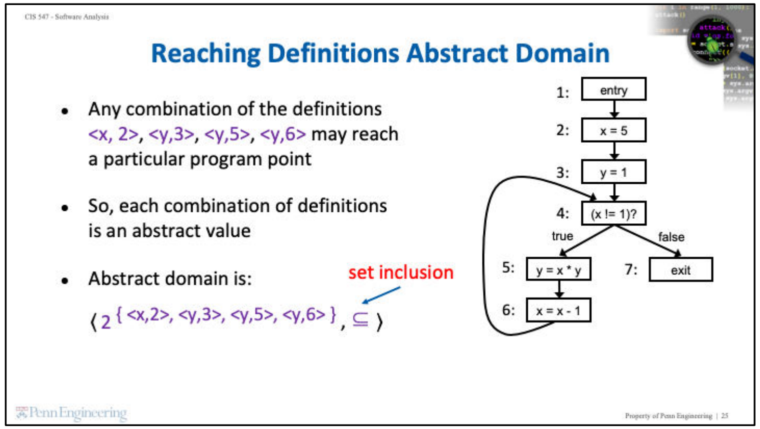
At this point, you might be wondering whether the chaotic iteration algorithm is guaranteed to terminate for every program in the WHILE language, despite its seemingly chaotic nature.

The answer, perhaps somewhat surprisingly, is yes.

The reason for this is two-fold. First, the two operations that this algorithm applies in each iteration are monotonic, that is, they never cause the IN and OUT sets to shrink.

Secondly, the largest that any such set can get is the set of all definitions in the program, which is finite. This implies that the IN and OUT sets cannot grow forever.

Together, these two reasons ensure that all IN and OUT sets will stop changing in some iteration of the chaotic iteration algorithm, which is the condition under which the algorithm terminates.



Reaching Definitions Abstract Domain

- Any combination of the definitions <x, 2>, <y,3>, <y,5>, <y,6> may reach a particular program point
- So, each combination of definitions is an abstract value
- Abstract domain is:

  $\langle 2^{\{<x,2>, <y,3>, <y,5>, <y,6>\}}, \subseteq \rangle$

set inclusion

To better understand how the IN and OUT sets evolve during the chaotic iteration algorithm, let's look at the abstract domain for reaching definitions analysis.

We introduced abstract domains in the first module of this course. The abstract domain defines the set of abstract values, say S, and a partial order <= on the set: an abstract value A is less than or equal to an abstract value B in the partial order if A is at least as precise as B *(Draw < S, <= > when you say this.)*

Let's see what constitutes an abstract value in reaching definitions analysis and then define its abstract domain.

Recall that reaching definitions analysis tracks the flow of definitions. There are four possible definitions in this program:

- the assignment to variable x at node 2,
- the assignment to variable y at node 3,
- the assignment to variable y at node 5, and
- The assignment to variable y at node 6.

Strictly speaking, we should also include two more definitions, one each capturing the fact that variables x and y are uninitialized at the start of the program. But we will omit these for the sake of brevity.

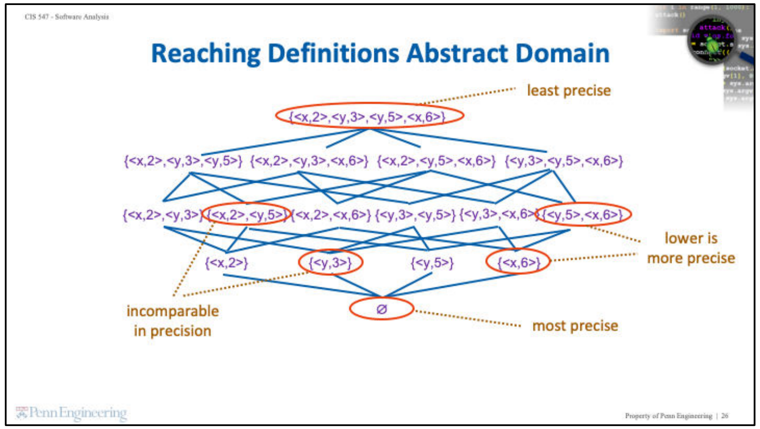
Any combination of these definitions may reach a particular program point.

Therefore, each combination of these definitions is an abstract value.

Since the abstract domain is the set of abstract values, it becomes the powerset of the set of these four definitions. We use the standard notation that the powerset of a set S is denoted by 2^S.

Furthermore, these abstract values are partially ordered by set inclusion, since a smaller set of reaching definitions is a more precise abstract value.

Let us visualize this abstract domain next.



CIS 547 - Software Analysis

Reaching Definitions Abstract Domain

least precise
{<x,2>,<y,3>,<y,5>,<x,6>}
{<x,2>,<y,3>,<y,5>} {<x,2>,<y,3>,<x,6>} {<x,2>,<y,5>,<x,6>} {<y,3>,<y,5>,<x,6>}
{<x,2>,<y,3>} {<x,2>,<y,5>} {<x,2>,<x,6>} {<y,3>,<y,5>} {<y,3>,<x,6>} {<y,5>,<x,6>}
lower is more precise
{<x,2>} {<y,3>} {<y,5>} {<x,6>}
incomparable in precision
∅
most precise

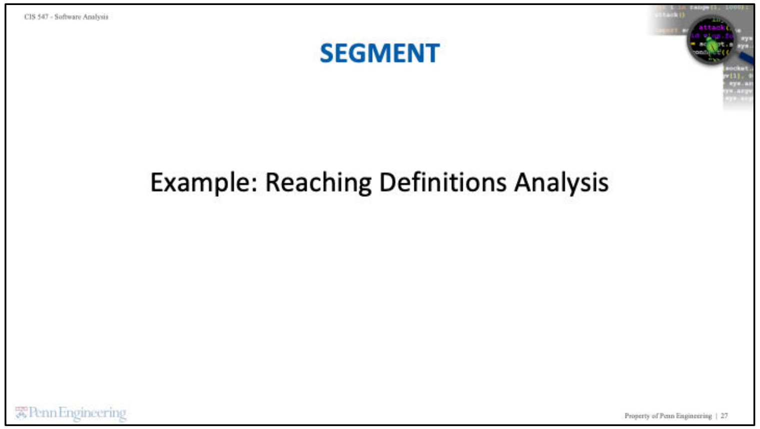
PennEngineering

Property of Penn Engineering | 26

The nodes in this structure denote abstract values, that is, sets of definitions, while the edges denote the set-inclusion partial order between them. The edges are directed in that they always go from a lower node to a higher node. The bottom node, the empty set of definitions, is the most precise, and the top node, the set of all definitions, is the least precise.

Also, note that this is a partial order because not all pairs of nodes are comparable in precision, in particular, those pairs of nodes that are not connected by a path, such as the set containing the definition of y at 3 and the set containing the definitions of x at 2 and y at 5. But the set containing the definition of x at 6 is strictly more precise than the set containing the definitions of x and 6 and y at 5.

At the start of the chaotic iteration algorithm, The IN and OUT sets of each node in the control-flow graph start out with the bottom abstract value, the empty set of definitions. In each iteration, these sets grow progressively, along the edges in this structure. The most that any IN or OUT set can grow is the top abstract value, the set of all definitions. Of course, we hope that most IN and OUT sets will saturate before that, thereby yielding a more precise analysis result. After all, we wouldn't need the chaotic iteration algorithm to conclude that all definitions might reach every node in the control-flow graph.



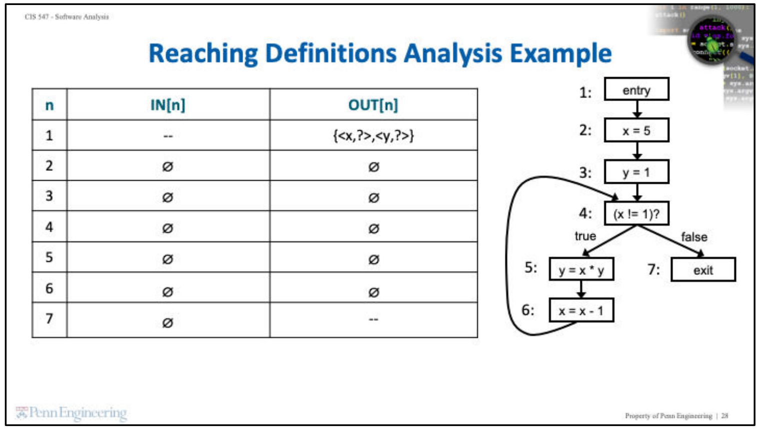
CIS 547 - Software Analysis

SEGMENT

Example: Reaching Definitions Analysis

Penn Engineering

Property of Penn Engineering | 27



CIS 547 - Software Analysis

Reaching Definitions Analysis Example

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	--

Penn Engineering

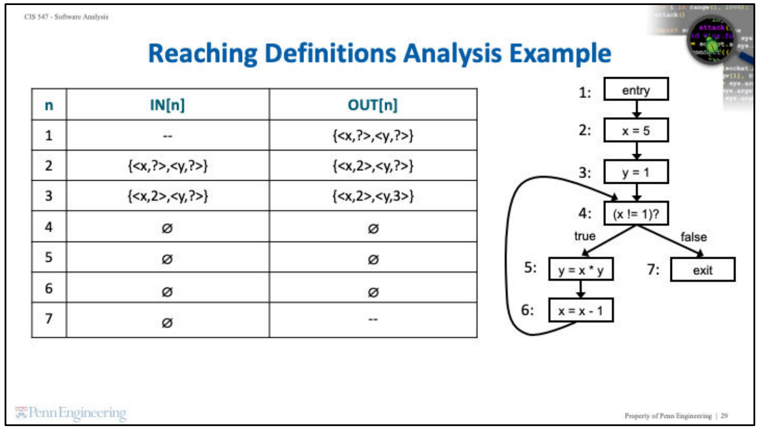
Property of Penn Engineering | 28

Now let’s see how the chaotic iteration algorithm for reaching definitions analysis works on our example control-flow graph.

We will use this table to track the values of the IN and OUT sets of each of the 7 nodes in this control-flow graph.

The algorithm starts by initializing all these entries to the empty set, except for the OUT set of the entry node, which captures the fact that both variables x and y are undefined at this point. Also, we will ignore the IN set of the entry node and the OUT set of the exit node as these nodes do not contain any statement and are merely placeholders.

Next, the algorithm repeatedly picks each of the remaining 5 nodes, numbered 2 thru 6, and applies the two rules that update their IN and OUT sets.



CIS 547 - Software Analysis

Reaching Definitions Analysis Example

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	--

Penn Engineering

Property of Penn Engineering | 29

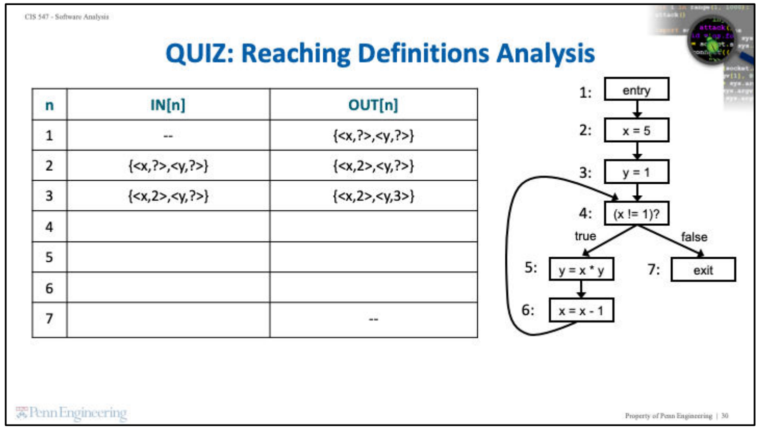
For instance, it updates the IN set of node 2 to reflect the fact that both variables x and y remain uninitialized at this point.
Strike out ∅ in IN column of row 2 and replace it with { <x,?>, <y,?> }.

It also updates the OUT set of node 2 to reflect the fact that the definition of variable x generated at node 2 reaches the exit of node 2. Notice that node 2 also kills the incoming fact that x is uniniitialized, but retains the incoming fact that y is uninitialized.
Strike out ∅ in OUT[2] and replace it with { <x,2>, <y,?> }.

Likewise, it updates the IN set of node 3 to reflect the fact that the definition of x at node 2 reaches it and the fact that y is still uninitialized.
Strike out ∅ in IN[3] and replace it with { <x,2>, <y,?> }.

Continuing further, it updates the OUT set of node 3 to reflect the fact that not only is the incoming definition of variable x not overwritten by node 3, but furthermore, node 3 generates a new definition of variable y. Both these definitions reach the exit of node 3.
Strike out ∅ in OUT[3] and replace it with { <x,2>, <y,3> }.

Let's update the remaining entries of this table in the following quiz.



CIS 547 - Software Analysis

QUIZ: Reaching Definitions Analysis

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}
4		
5		
6		
7		--

Penn Engineering

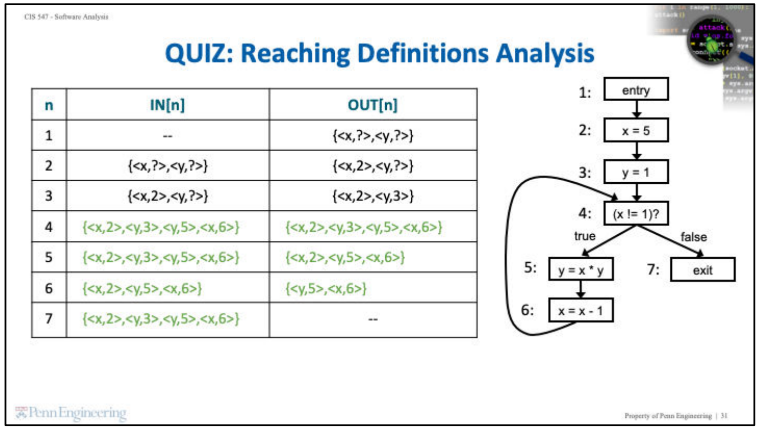
Property of Penn Engineering | 30

{QUIZ SLIDE}

Fill in the blank entries in the table with the final values of the corresponding IN and OUT sets computed by the chaotic iteration algorithm for reaching definitions analysis.

Keep in mind the two operations that the algorithm applies to each node. The first operation states that the IN set of a node is the union of the OUT sets of its immediate predecessor nodes. The second operation states that the OUT set of a node contains all the definitions in the IN set of that node, minus definitions that are overwritten by that node, plus any new definitions that are generated by that node.

Remember to keep iterating through the whole table, applying these two operations to each node until there are no changes in a given pass through the table.



CIS 547 - Software Analysis

QUIZ: Reaching Definitions Analysis

n	IN[n]	OUT[n]
1	--	{<x,?>,<y,?>}
2	{<x,?>,<y,?>}	{<x,2>,<y,?>}
3	{<x,2>,<y,?>}	{<x,2>,<y,3>}
4	{<x,2>,<y,3>,<y,5>,<x,6>}	{<x,2>,<y,3>,<y,5>,<x,6>}
5	{<x,2>,<y,3>,<y,5>,<x,6>}	{<x,2>,<y,5>,<x,6>}
6	{<x,2>,<y,5>,<x,6>}	{<y,5>,<x,6>}
7	{<x,2>,<y,3>,<y,5>,<x,6>}	--

Penn Engineering

Property of Penn Engineering | 31

{SOLUTION SLIDE}

Let's review the solution.

Consider the condition node labeled 4. Its IN set consists of all facts from the OUT sets of its immediate predecessors, which are nodes 3 and 6. So the IN set contains the definition of x at node 2 and the definition of y at node 3 (write <x,2> and <y,3> in IN[4]). The condition node neither generates nor overwrites any definitions, so we copy these two facts to the OUT set (write <x,2> and <y,3> in OUT[4]).

Next, we copy the OUT set of node 4 to the IN set of node 5 (write <x,2> and <y,3> in IN[5]).

Now, to compute OUT[5], we first copy <x,2> and <y,3> from IN[5] (write <x,2> and <y,3> in OUT[5]). Since the node overwrites the definition of y, we delete <y,3> and add <y,5> (erase <y,3> and write <y,5> in OUT[5]).

Since node 5 is the only immediate predecessor of node 6, we copy OUT[5] to IN[6] (write <x,2> and <y,5> in IN[6]).

Then, at node 6, we first copy IN[6] to OUT[6] (write <x,2> and <y,5> in OUT[6]). Since node 6 redefines x, we delete <x,2> and replace it with <x,6> (erase <x,2> and write <x,6> in IN[6]).

Finally, we look at the immediate predecessor of node 7, which is node 4, to determine the IN set of node 7 (write <x,2> and <y,3> in IN[7]).

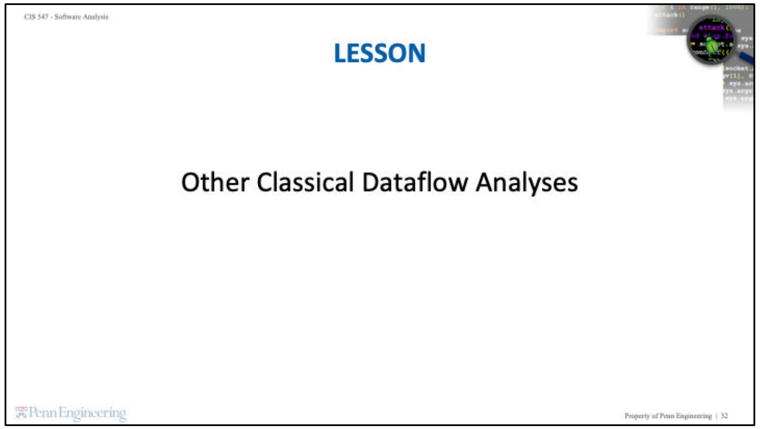
Now we loop back to our earlier condition node 4. We've already visited this point, so we already have an IN set. Therefore we'll union the existing IN set with the OUT of the newly found predecessor, node 6. (write <y,5> and <x,6> in IN[4]) This gives us the new set: <x,2>,<x,6>,<y,3> ,<y,5>. This will in turn modify our OUT to be <x,2>,<x,6>,<y,3> ,<y,5> as well (write <y,5> and <x,6> in OUT[4]).

This change propagates down through the left branch, updating the IN and OUT sets of node 5 and the IN set of node 6 (write <y,5> and <x,6> in IN[5]; write <x,6> in OUT[5]; write <x,6> in IN[6]).

Now, looking at our last node, the exit, we look at the OUT for node 4 and take that as our IN set (write <y,5> and <x,6> in IN[7]).

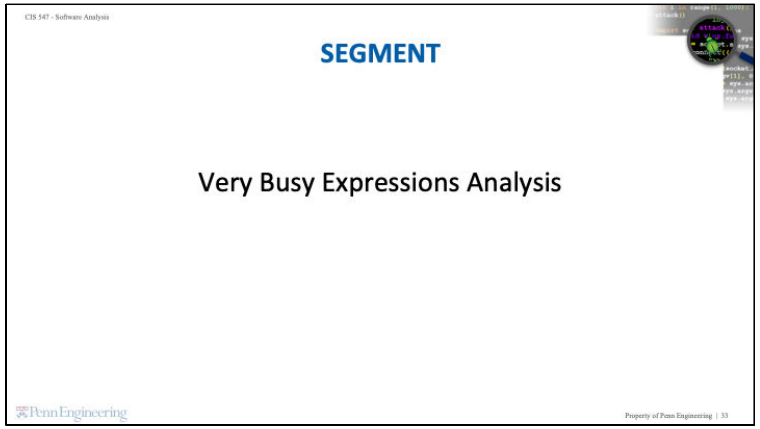
Finally, making another pass through the table, we see that there are no more updates made to the IN or OUT sets of any node. This means we're done!

Recall that one use of reaching definitions analysis was to find uses of uninitialized variables. A variable used at a node, such as the variable x at node 4, may be uninitialized if the hypothetical definition of x at the program entry is contained in IN[4]. Since IN[4] does not contain this definition, we have proven that x will always refer to a properly initialized value whenever the conditional x != 1 at node 4 is evaluated. As an exercise, repeat the analysis this time assuming that the program does not contain the definition of x at node 2. In that case, you will see that the hypothetical definition of x at the program entry is contained in IN[4].



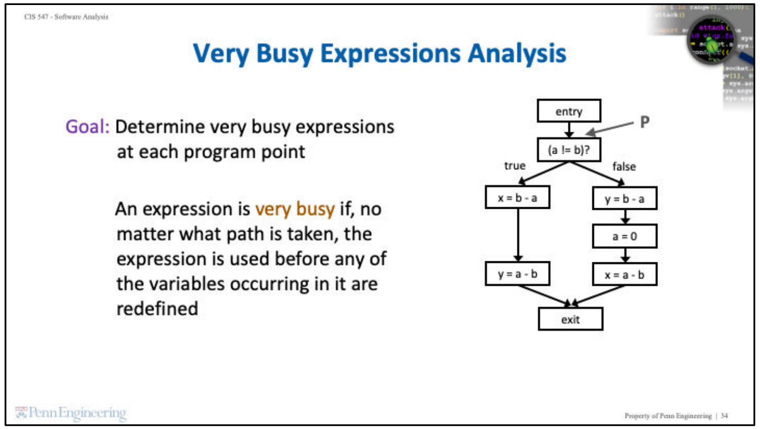
LESSON

Other Classical Dataflow Analyses



SEGMENT

Very Busy Expressions Analysis



CIS 547 - Software Analysis

Very Busy Expressions Analysis

Goal: Determine very busy expressions at each program point

An expression is very busy if, no matter what path is taken, the expression is used before any of the variables occurring in it are redefined

PennEngineering

Property of Penn Engineering | 34

Now let's move on to the second of the four classical dataflow analyses. This one is called Very Busy Expressions analysis. We will present this analysis by following a recipe similar to the one we used to present Reaching Definitions analysis.

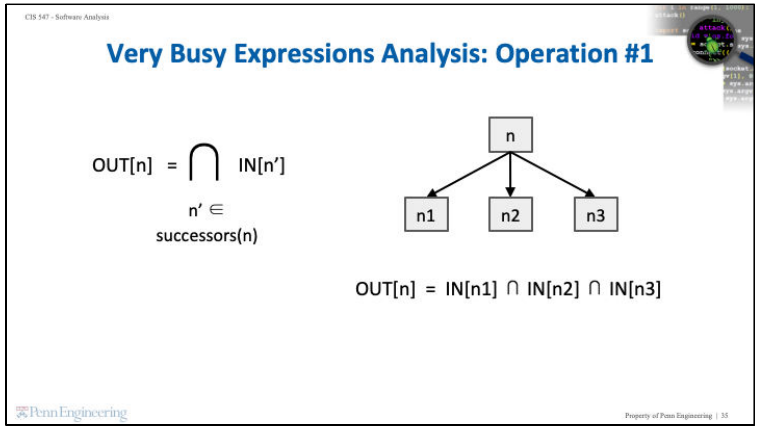
The goal of Very Busy Expressions analysis is to compute expressions that are very busy at each program point.

An expression is very busy if, no matter what path is taken, the expression is always used before any of the variables occurring in it are redefined.

Let us look at the following example program. Let's consider these two expressions in this program: a - b and b - a.

Consider the program point P at the entry of the condition statement.

The expression b - a is very busy at this point since it is used along both the paths from that point, here and here (gesture), before any of the variables occurring in the expression is redefined. Now let's consider expression a - b. This expression is used on both the paths as well, but variable a in the expression is redefined along one of those paths here (gesture), before the expression is used here (gesture). So expression a - b is not very busy at program point P.



CIS 547 - Software Analysis

Very Busy Expressions Analysis: Operation #1

$$\mathrm{OUT}[n] \;=\; \bigcap_{n' \in \,\mathrm{successors}(n)} \mathrm{IN}[n']$$

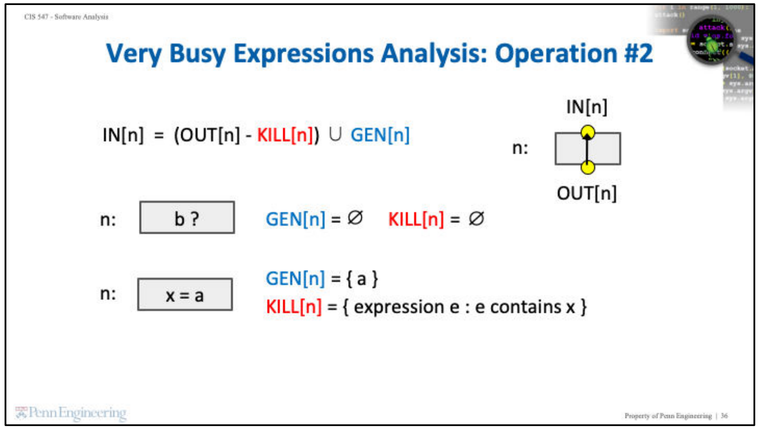
OUT[n] = IN[n1] ∩ IN[n2] ∩ IN[n3]

PennEngineering

Property of Penn Engineering | 35

Here's the first of the two operations of very-busy-expressions analysis. This rule states how to compute the set of facts at the exit of a particular node in the control-flow graph. We do this by taking the intersection of the sets of expressions at the entry of that node's immediate successors: that is, the intersection of IN[n'] for each successor node n' of n.

For instance, suppose node n has three immediate successors n1, n2, and n3. Then, the set of facts at the exit of n is the intersection of the sets of facts at the entry of n1, n2, and n3.



CIS 547 - Software Analysis

Very Busy Expressions Analysis: Operation #2

IN[n] = (OUT[n] - KILL[n]) ∪ GEN[n]

n: b ? GEN[n] = ∅ KILL[n] = ∅

n: x = a GEN[n] = { a }
KILL[n] = { expression e : e contains x }

Property of Penn Engineering | 36

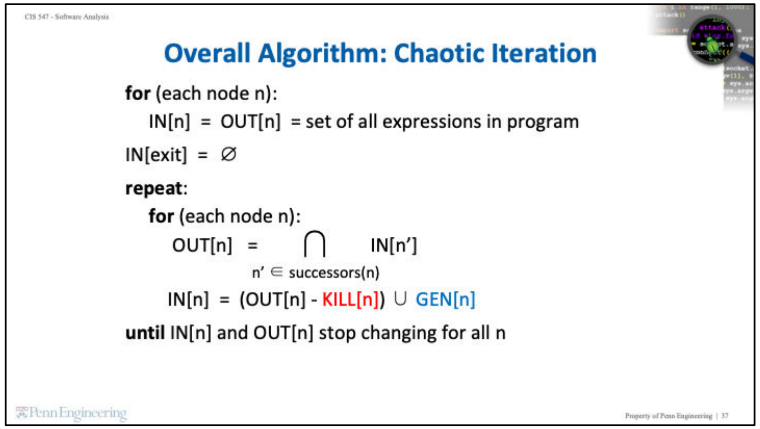
The second operation of very-busy-expressions analysis tells us how to compute the set of facts at the entry of a particular node from the set of facts at the exit of that node. The operation of this rule depends on the statement that occurs at the node we are looking at.

Like we did for reaching-definitions analysis, we'll first state this operation in its general form, and then we'll show specific instances of the operation corresponding to each kind of primitive statement in the WHILE language.

The general form of this operation states that the set of expressions at the entry of node n is equal to the set of expressions at the exit of node n, minus any expressions using variables that are modified by node n, unioned with any new expressions that are used by node n. Like before, we call these sets of expressions the KILL set and the GEN set.

If the statement at node n is a conditional statement, then the KILL set and GEN set are empty, since nothing is modified or generated by the node.

If the statement at node n is an assignment statement of the form x = a, then the GEN set will contain the expression assigned to variable x at node n itself, reflecting the fact that this expression is used by node n. The KILL set, on the other hand, contains every expression using x, reflecting the fact that all those expressions will be modified by node n.



CIS 547 - Software Analysis

Overall Algorithm: Chaotic Iteration

```
for (each node n):
  IN[n] = OUT[n] = set of all expressions in program
IN[exit] = ∅
repeat:
  for (each node n):
    OUT[n] = ⋂_{n' ∈ successors(n)} IN[n']
    IN[n] = (OUT[n] - KILL[n]) ∪ GEN[n]
until IN[n] and OUT[n] stop changing for all n
```

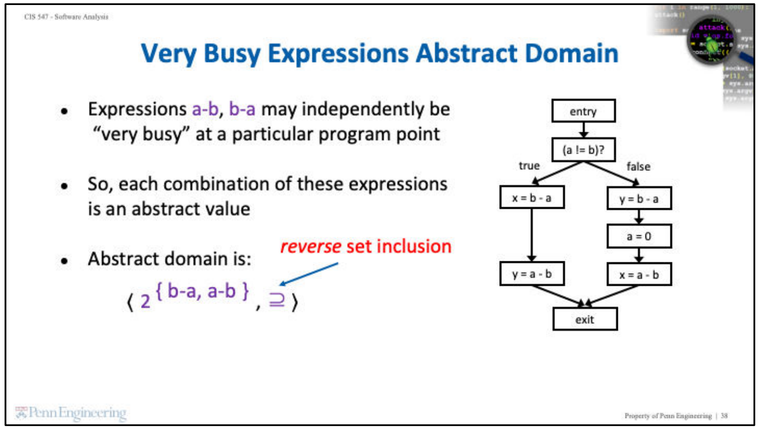
Property of Penn Engineering | 37

The overall algorithm for very busy expressions analysis is nearly identical to that for reaching definitions analysis but has three notable differences.

First, the two operations that are applied in each iteration of the algorithm are slightly different. Notice that the roles of IN and OUT sets in these two operations are switched, reflecting the fact that very-busy-expressions analysis propagates information backwards in a control-flow graph, in contrast to reaching-definitions analysis, which propagates information forward.

Moreover, we take the intersection of sets as opposed to their union, which causes the IN and OUT sets of very busy expressions to shrink as the algorithm progresses. In contrast, recall that in reaching-definitions analysis, the IN and OUT sets of reaching definitions grow because we use the union operation.

This also makes it clear why we initialize all IN and OUT sets in very busy expressions analysis to the set of all expressions in the program, since these sets will shrink as the algorithm progresses. In contrast, recall that the IN and OUT sets in reaching definitions analysis are initialized to the empty set, and those sets grow as that algorithm progresses. (However, we still set the IN set of the exit node of the control-flow graph to be empty, since no expressions are very busy at the end of the program.)



Very Busy Expressions Abstract Domain

- Expressions a-b, b-a may independently be "very busy" at a particular program point
- So, each combination of these expressions is an abstract value
- Abstract domain is: $\langle 2^{\{\text{b-a, a-b}\}}, \supseteq \rangle$

reverse set inclusion

entry
(a != b)?
true
false
x = b - a
y = b - a
a = 0
y = a - b
x = a - b
exit

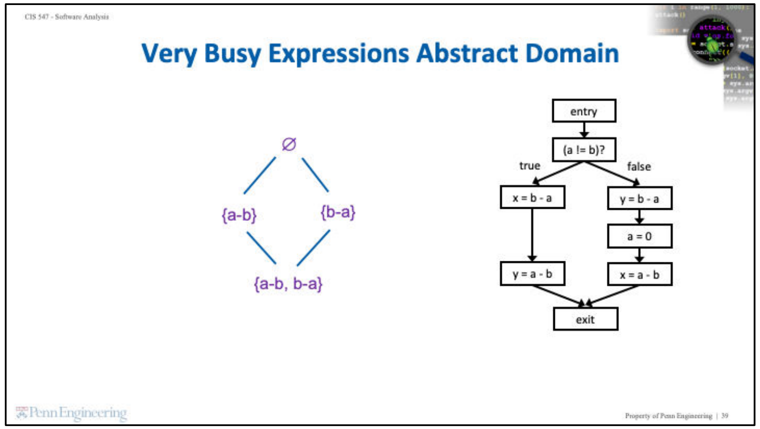
Now, let's consider the abstract domain for very busy expressions analysis.

The expressions a – b and b – a may independently be very busy at a particular program point.

So, each combination of these two expressions is an abstract value.

Since the abstract domain is the set of abstract values, it becomes the powerset of the set of these two expressions.

Furthermore, these abstract values are partially ordered by reverse set inclusion, since a larger set of very busy expressions corresponds to a more precise abstract value. In contrast, abstract values in reaching definitions analysis were ordered by set inclusion, since a smaller set of reaching definitions corresponded to a more precise abstract value.



Very Busy Expressions Abstract Domain

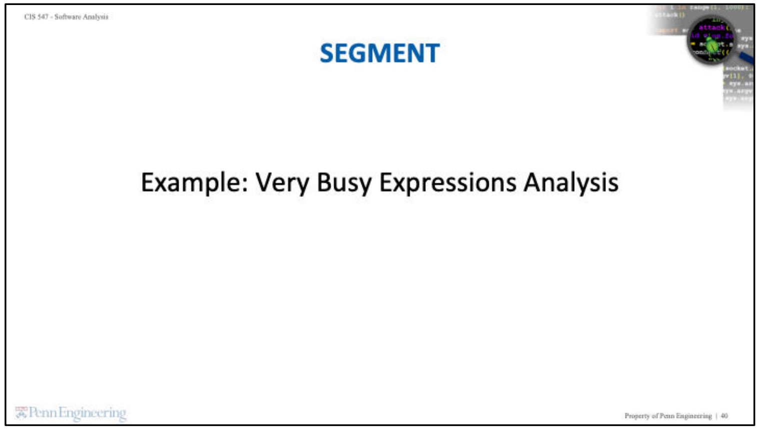
∅
{a-b}
{b-a}
{a-b, b-a}

entry
(a != b)?
true
false
x = b - a
y = b - a
a = 0
y = a - b
x = a - b
exit

Here is a visualization of the abstract domain for very-busy-expressions analysis on our example program.

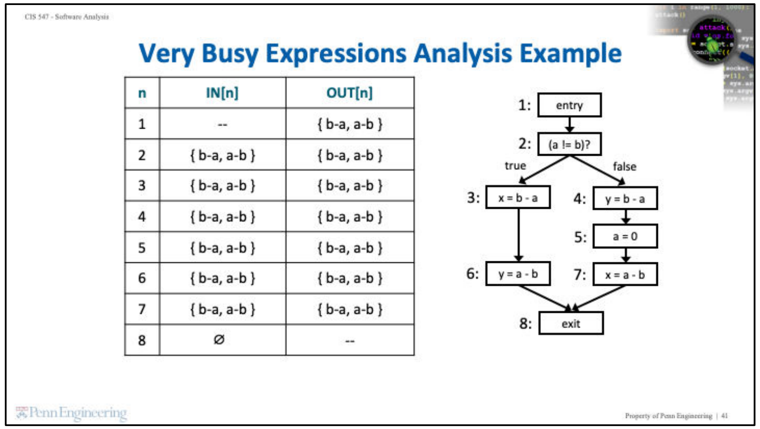
Again, the nodes are abstract values, and the edges represent the reverse-set-inclusion partial order between them. The bottom node, the set of all expressions, is the most precise, and the top node, the empty set of expressions, is the least precise.

The IN and OUT sets of each node in the control-flow graph start out with the bottom abstract value, the set of all expressions, at the start of the chaotic iteration algorithm. In each iteration, these sets shrink progressively, along the edges in this structure. The most that any IN or OUT set can shrink is the top abstract value, the empty set of expressions.



SEGMENT

Example: Very Busy Expressions Analysis



Very Busy Expressions Analysis Example

n	IN[n]	OUT[n]
1	--	{ b-a, a-b }
2	{ b-a, a-b }	{ b-a, a-b }
3	{ b-a, a-b }	{ b-a, a-b }
4	{ b-a, a-b }	{ b-a, a-b }
5	{ b-a, a-b }	{ b-a, a-b }
6	{ b-a, a-b }	{ b-a, a-b }
7	{ b-a, a-b }	{ b-a, a-b }
8	∅	--

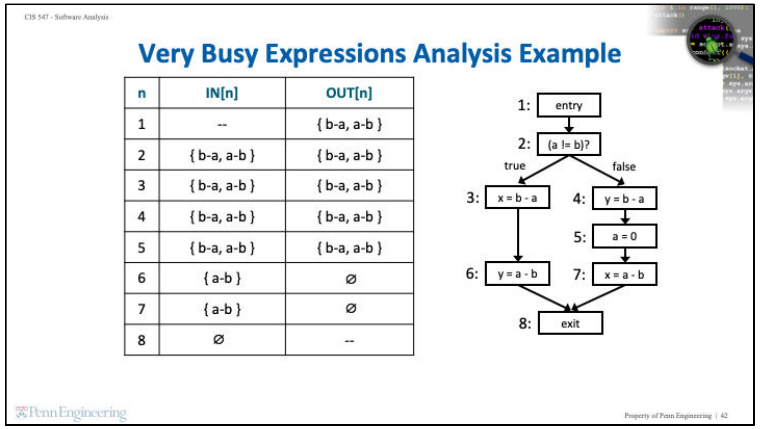
Now let's see how the chaotic iteration algorithm for very-busy-expressions analysis works on our example control-flow graph.

We will use this table to track the values of the IN and OUT sets of each of the 8 nodes in this control-flow graph.

The algorithm starts by initializing all these entries to the set of all expressions, except the OUT set of the exit node, which it initializes to the empty set.

As in the case of reaching definitions analysis, we will ignore the IN set of the entry node and the OUT set of the exit node as these nodes do not contain any statement and are merely placeholders.

Next, the algorithm repeatedly picks each of the remaining nodes and applies the two rules that update their IN and OUT sets.



Very Busy Expressions Analysis Example

n	IN[n]	OUT[n]
1	--	{ b-a, a-b }
2	{ b-a, a-b }	{ b-a, a-b }
3	{ b-a, a-b }	{ b-a, a-b }
4	{ b-a, a-b }	{ b-a, a-b }
5	{ b-a, a-b }	{ b-a, a-b }
6	{ a-b }	∅
7	{ a-b }	∅
8	∅	--

1: entry
2: (a != b)?
true
false
3: x = b - a
4: y = b - a
5: a = 0
6: y = a - b
7: x = a - b
8: exit

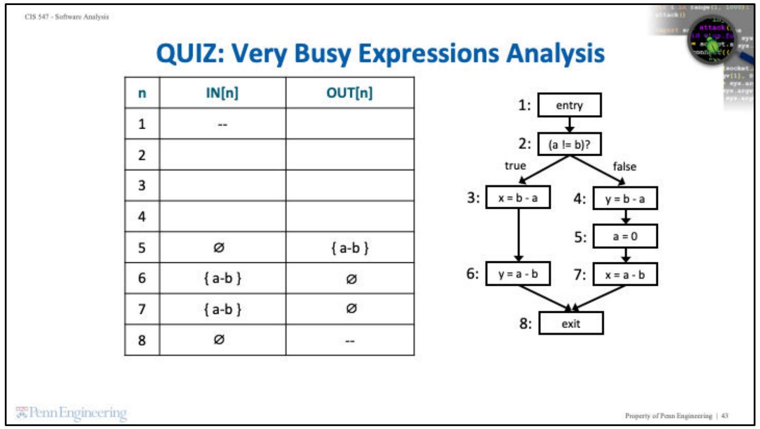
For instance, it updates the OUT set of node 7 to the empty set, reflecting the fact that no expressions are very busy at this point.
Erase { b-a, a-b } in OUT[7] and replace it with ∅.

Likewise, it updates the IN set of node 7 to reflect the fact that expression a - b is very busy at this point, as it is indeed used immediately thereafter.
Erase { b-a, a-b } in IN[7] and replace it with { a-b }.

Continuing along this branch, it similarly updates the OUT set of node 6 to the empty set, as no expressions are very busy at this point.
Erase { b-a, a-b } in OUT[6] and replace it with ∅.

Similarly, it continues further backwards and updates the IN set of node 6 to reflect the fact that expression a - b is very busy at this point.
Erase { b-a, a-b } in IN[6] and replace it with { a-b }.

Let's update the remaining entries of this table in the following quiz.



QUIZ: Very Busy Expressions Analysis

n	IN[n]	OUT[n]
1	--	
2		
3		
4		
5	∅	{ a-b }
6	{ a-b }	∅
7	{ a-b }	∅
8	∅	--

1: entry
2: (a != b)?
true
false
3: x = b - a
4: y = b - a
5: a = 0
6: y = a - b
7: x = a - b
8: exit

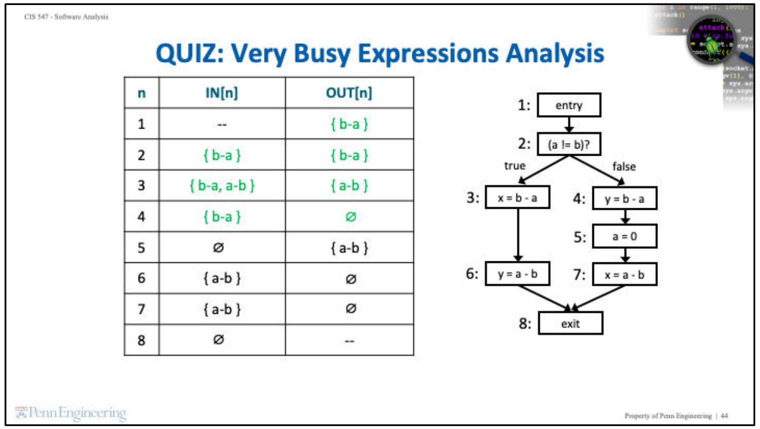
{QUIZ SLIDE}

Fill in the blank entries in the table with the final values of the corresponding IN and OUT sets computed by the chaotic iteration algorithm for very busy expressions analysis.

Keep in mind the two operations that the algorithm applies to each node.

The first operation states that the OUT set of a node is the intersection of the IN sets of its immediate successor nodes. The second operation states that the IN set of a node contains all the expressions in the OUT set of that node, minus expressions that are overwritten by that node, plus any new expressions that are generated by that node.

Remember to keep iterating through the whole table, applying these two operations to each node until there are no changes in a given pass through the table.



CIS 547 - Software Analysis

QUIZ: Very Busy Expressions Analysis

n	IN[n]	OUT[n]
1	--	{ b-a }
2	{ b-a }	{ b-a }
3	{ b-a, a-b }	{ a-b }
4	{ b-a }	∅
5	∅	{ a-b }
6	{ a-b }	∅
7	{ a-b }	∅
8	∅	--

Penn Engineering

Property of Penn Engineering | 44

{SOLUTION SLIDE}

Let's trace the chaotic iteration algorithm through until it completes its iteration. Note that since we don't have any loops in this program, we will only need to make one pass upwards before the IN and OUT sets stabilize.

Let's look at node 4. Its only successor is node 5, so we copy IN[5] to OUT[4] (replace OUT[4] by the empty set). Since OUT[4] is empty, we have no expressions to kill. All we need to do is add b-a to the set since the expression is being used at this node (replace IN[4] by the set {b-a}).

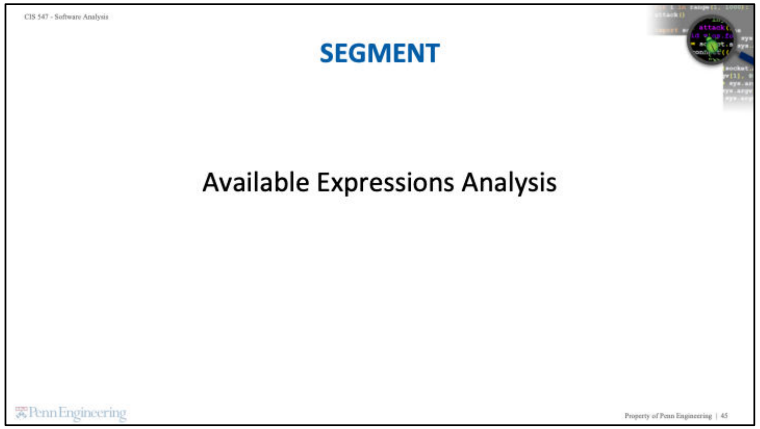
For node 3, our OUT set is the same as the IN set of its sole successor, node 6 (replace OUT[3] by {a-b}).
Since x is not present in any expressions in the OUT set of node 3, we don't kill any expressions. We'll just add the expression being defined to the IN set (replace IN[3] by the set {a-b, b-a}).

Now, for node 2, we have two different successors we must use to determine our OUT set. We need to make sure that any expression we set as very busy is such for all execution paths. Thus, we take the intersection of program point 3 and 4's IN sets and get just b-a (replace OUT[2] by the set {b-a}). Since this node evaluates a boolean expression, we neither kill nor generate any expressions. We just copy the OUT set to the IN set (replace IN[2] by {b-a}).

Finally we reach the entry, whose OUT set is the same as the IN set of its only successor (replace OUT[1] by the set {b-a}).

Since there are no loops in the program, repeating the chaotic iteration algorithm will not cause any of the IN or OUT sets to change, so this completes our analysis.

As an exercise, think how a compiler might transform this program into one with smaller code size using the result of this analysis.



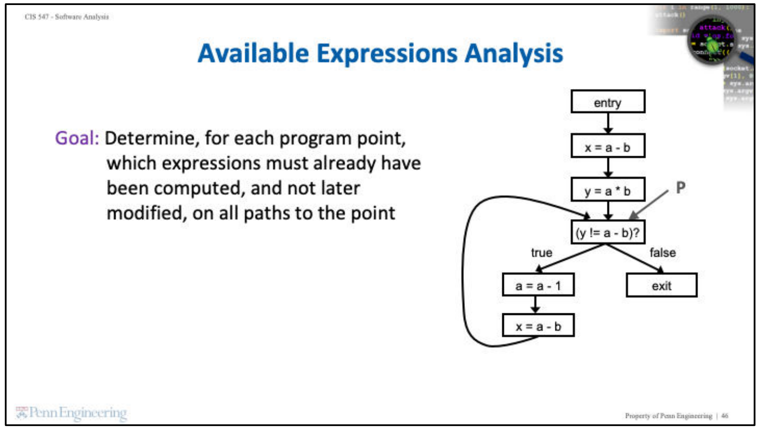
CIS 547 - Software Analysis

SEGMENT

Available Expressions Analysis

Penn Engineering

Property of Penn Engineering | 45



CIS 547 - Software Analysis

Available Expressions Analysis

Goal: Determine, for each program point, which expressions must already have been computed, and not later modified, on all paths to the point

Penn Engineering

Property of Penn Engineering | 46

Next, let’s move on to the third of our four classical dataflow analyses, called Available Expressions Analysis.

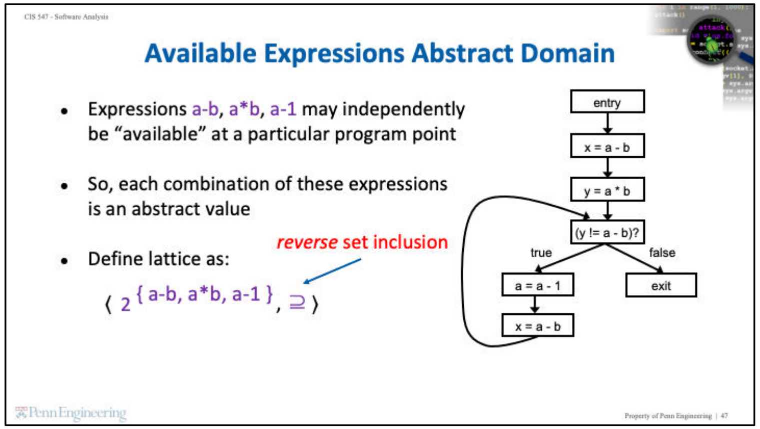
The goal of this analysis is to determine, for each program point, which expressions have already been computed, and not later modified, on all paths to the program point.

Let us look at the following example program. There are three expressions of interest in this program: a - b, a * b, and a - 1.

Consider the program point P at the entry of this condition.

At this point, the expression a - b is said to be available. To see why, let’s show that this expression is already computed and not later modified along every path that reaches this point. There are two paths. Along this path (gesture), the expression a - b is computed here (gesture), and neither a nor b is modified later. Likewise, along this other path (gesture), the expression a - b is computed here (gesture), and neither a nor b is modified later.

On the other hand, the expression a * b is not available at program point P. This is because, although this expression is available along this path (gesture), it is not available along this other path (gesture). In particular, the variable a occurring in this expression is modified here along this path (gesture), and the expression is not computed after this modification in order to become available later at point P.



Available Expressions Abstract Domain

- Expressions a-b, a*b, a-1 may independently be "available" at a particular program point
- So, each combination of these expressions is an abstract value
- Define lattice as:

reverse set inclusion

$\langle\, 2^{\{\text{a-b, a*b, a-1}\}},\ \supseteq\, \rangle$

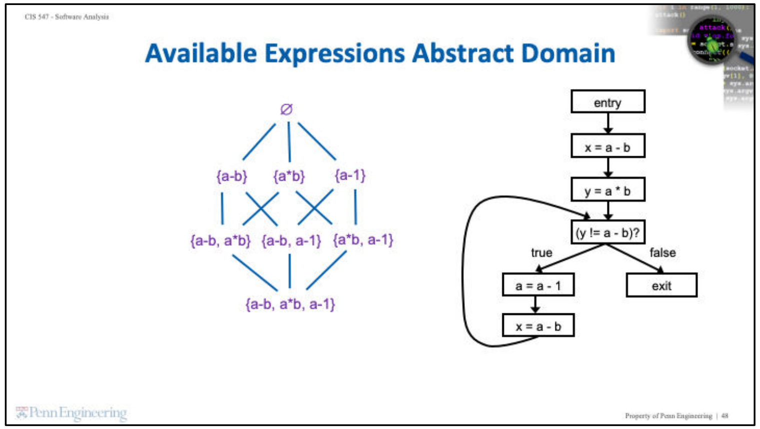
Now, let's consider the abstract domain for available expressions analysis.

The expressions a – b, a * b, and a – 1 may independently be available at a particular program point.

So, each combination of these three expressions is an abstract value.

Since the abstract domain is the set of abstract values, it becomes the powerset of the set of these three expressions.

Furthermore, these abstract values are partially ordered by reverse set inclusion, since a larger set of available expressions corresponds to a more precise abstract value. This is similar to what we saw for very busy expressions analysis.

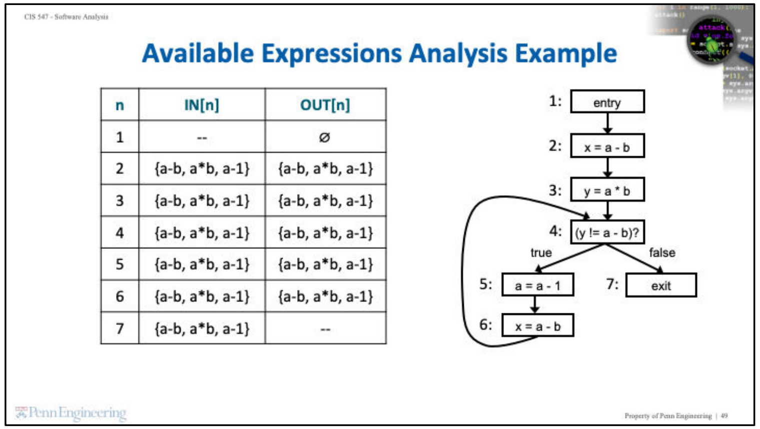


Available Expressions Abstract Domain

Here is a visualization of the abstract domain for available expressions analysis on our example program.

Again, the nodes are abstract values, and the edges represent the reverse-set-inclusion partial order between them. The bottom node, the set of all expressions, is the most precise, and the top node, the empty set of expressions, is the least precise.

At the start of the chaotic iteration algorithm, the IN and OUT sets of each node in the control-flow graph start out with the bottom abstract value, the set of all expressions. In each iteration, these sets shrink progressively, along the edges in this structure. The most that any IN or OUT set can shrink is the top abstract value, the empty set of expressions.

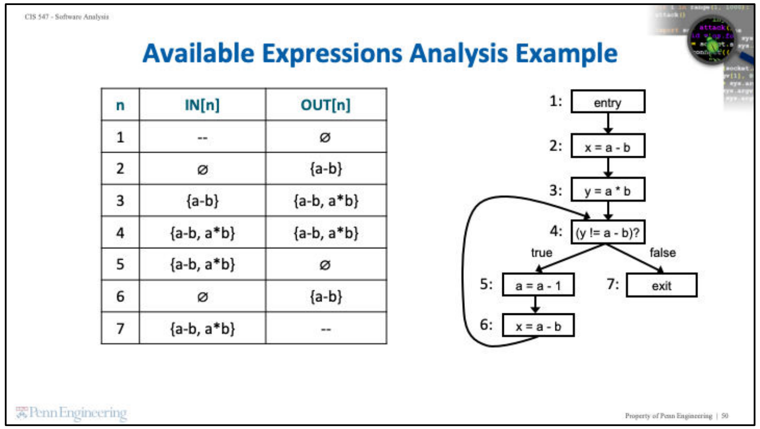


Available Expressions Analysis Example

n	IN[n]	OUT[n]
1	--	∅
2	{a-b, a*b, a-1}	{a-b, a*b, a-1}
3	{a-b, a*b, a-1}	{a-b, a*b, a-1}
4	{a-b, a*b, a-1}	{a-b, a*b, a-1}
5	{a-b, a*b, a-1}	{a-b, a*b, a-1}
6	{a-b, a*b, a-1}	{a-b, a*b, a-1}
7	{a-b, a*b, a-1}	--

1: entry
2: x = a - b
3: y = a * b
4: (y != a - b)?
true
false
5: a = a - 1
7: exit
6: x = a - b

Let’s walk through an example. Our IN set in this case will be any expressions which have been calculated earlier in the code without having the variables in their calculations overwritten. Our OUT set will be our IN set minus any expressions which have a variable that is overwritten by that statement, plus any expressions that are generated by that statement.



Available Expressions Analysis Example

n	IN[n]	OUT[n]
1	--	∅
2	∅	{a-b}
3	{a-b}	{a-b, a*b}
4	{a-b, a*b}	{a-b, a*b}
5	{a-b, a*b}	∅
6	∅	{a-b}
7	{a-b, a*b}	--

1: entry
2: x = a - b
3: y = a * b
4: (y != a - b)?
true
false
5: a = a - 1
7: exit
6: x = a - b

Let’s walk through the first three program points. For the entry, we have the empty set as our OUT set (gesture to OUT[1]).

This OUT set will be copied to the IN set of node 2 (write ∅ in IN[2]). To compute the OUT set of node 2, we first copy the IN set of node 2, which is empty, and add expression a-b which is generated at node 2 (write {a-b} in OUT[2]).

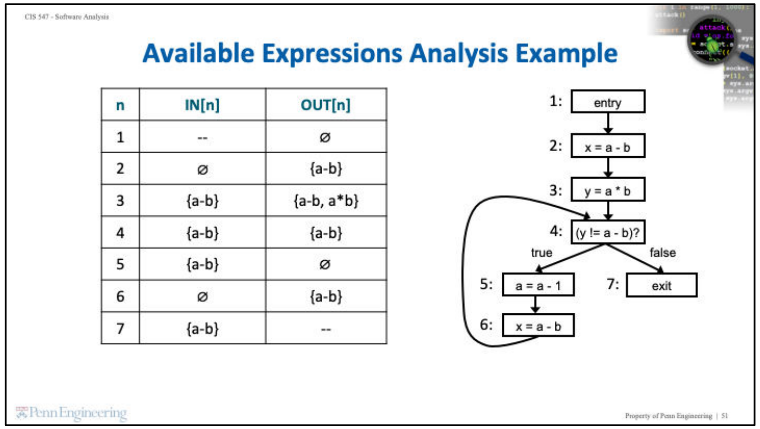
Node 3 takes in the expression a-b from node 2 (write {a-b} in IN[3]). In addition, node 3 generates another expression, a*b, which we add to its OUT set (write {a-b, a*b} in OUT[3]).

Node 4 is a bit tricky. Our IN set at this point is a-b and a*b (write {a-b, a*b} in IN[4]). Note that this is not the only path to this program point, but we have not computed the OUT set of node 4’s other predecessor yet, so we’ll need to make at least another pass through the table before the algorithm stabilizes. Nothing is killed or generated at this node, so we copy the IN set to the OUT set (write {a-b, a*b} in OUT[4]).

At node 5, we first copy the OUT set of node 4 to the IN set (write {a-b, a*b} in IN[5]). Node 5 seems to generate a-1 but in fact it does not, since it immediately overwrites a. Also, we’ll need to kill each expression in our IN set that uses a. That’s expressions a-b and a*b. So we’re left with an empty OUT set, which we also copy over to the IN set of node 6 (write ∅ in OUT[5] and IN[6]).

At node 6 we generate a-b once again, and so our new OUT set is a-b (write {a-b} in OUT[6]).

At node 7, we copy OUT[4] to IN[7] (write {a-b, a*b} in IN[7]) at which point we've completed our first pass through the table.



CIS 547 - Software Analysis

Available Expressions Analysis Example

n	IN[n]	OUT[n]
1	--	∅
2	∅	{a-b}
3	{a-b}	{a-b, a*b}
4	{a-b}	{a-b}
5	{a-b}	∅
6	∅	{a-b}
7	{a-b}	--

Penn Engineering

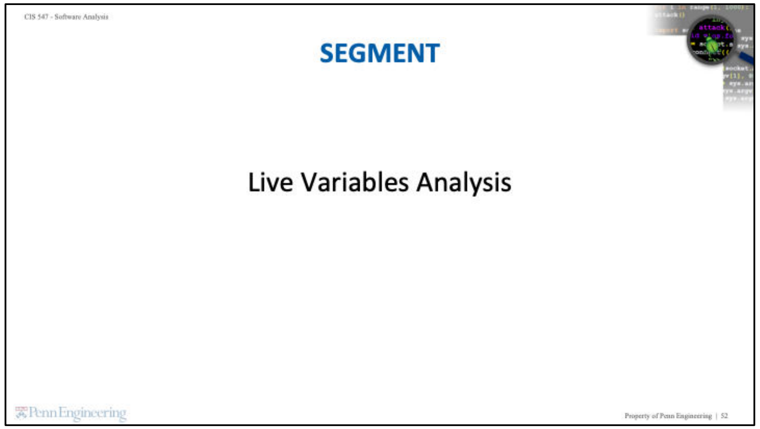
Property of Penn Engineering | 51

In the next pass, we revisit node 4. We need to make sure that the expressions in IN[4] are available on all program paths to this point, so we take the intersection of OUT[3] with OUT[6], leaving just a-b (replace IN[4] by {a-b}). This reflects the fact that the expression a * b is not in fact available at this point.

Passing through the remaining nodes after node 4, OUT[4], IN[5], and IN[7] become just a-b (replace all of these by {a-b}).

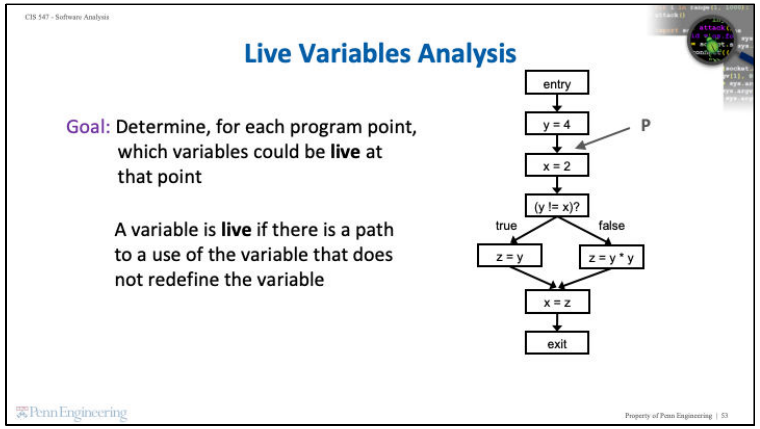
No more changes will be made by additional iterations, so this completes the analysis.

As an exercise, think how a compiler might transform this program into one which avoids recomputing an expression using the result of this analysis.



SEGMENT

Live Variables Analysis



Live Variables Analysis

Goal: Determine, for each program point, which variables could be **live** at that point

A variable is **live** if there is a path to a use of the variable that does not redefine the variable

entry

y = 4

P

x = 2

(y != x)?

true

false

z = y

z = y * y

x = z

exit

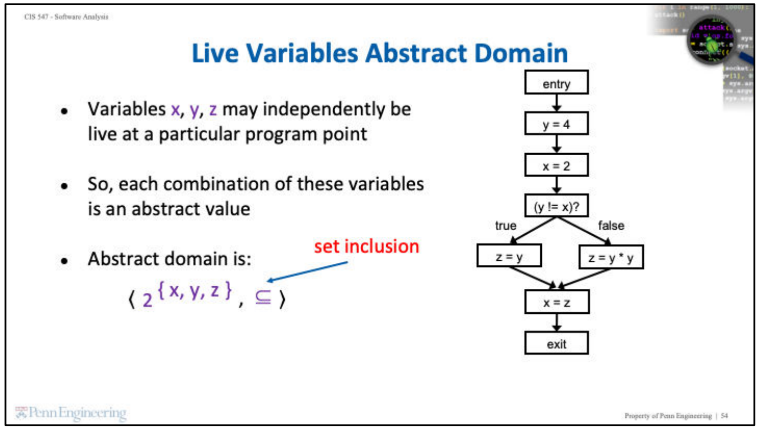
Now let's move on to the final of our four classical dataflow analyses, called Live Variables Analysis.

The goal of live variables analysis is to determine for each program point, which variables may be live at the point, where a variable is live if there is a path to a use of the variable that does not re-define the variable.

Let us look at the following example program. There are three variables in this program: x, y, and z.

Consider the program point P at the entry of this assignment to x (draw P and the arrow).

The variable y is live here because there is a path to a use of y here (gesture) that does not re-define y. The variable x, on the other hand, is not live at program point P because, even though there is a path to a use of x here (gesture), that path redefines x here (gesture). The variable z is also not live at program point P because along each of these paths emanating from P, z is redefined here and here (gesture) before it is used here (gesture).



CIS 547 - Software Analysis

Live Variables Abstract Domain

- Variables x, y, z may independently be live at a particular program point
- So, each combination of these variables is an abstract value
- Abstract domain is:

$\langle 2^{\{x, y, z\}}, \subseteq \rangle$

set inclusion

Penn Engineering

Property of Penn Engineering | 54

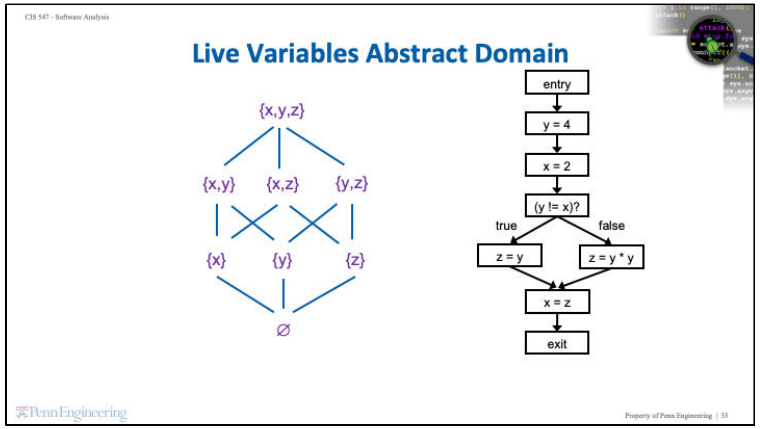
Now, let's consider the abstract domain for live variables analysis.

The variables x, y, and z may independently be live at a particular program point.

So, each combination of these three variables is an abstract value.

Since the abstract domain is the set of abstract values, it becomes the powerset of the set of these three variables.

Furthermore, these abstract values are partially ordered by set inclusion, since a smaller set of live variables corresponds to a more precise abstract value. This is similar to what we saw for reaching definitions analysis.



CIS 547 - Software Analysis

Live Variables Abstract Domain

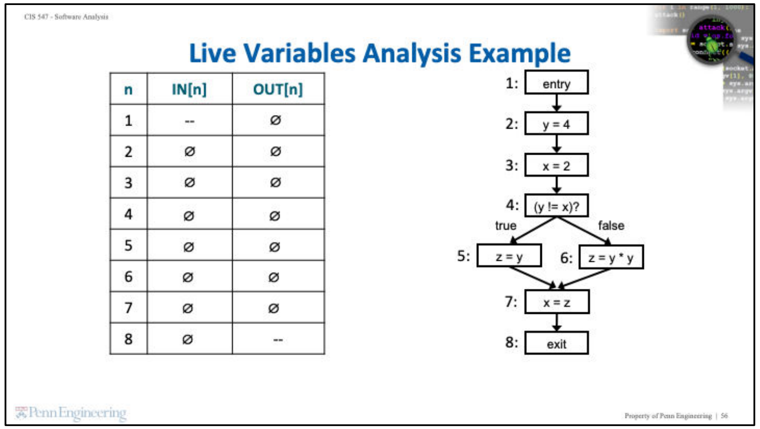
Penn Engineering

Property of Penn Engineering | 55

Here is a visualization of the abstract domain for live variables analysis on our example program.

Again, the nodes are abstract values, and the edges represent the set-inclusion partial order between them. The bottom node, the empty set of variables, is the most precise, and the top node, the set of all variables, is the least precise.

At the start of the chaotic iteration algorithm, the IN and OUT sets of each node in the control-flow graph start out with the bottom abstract value, the empty set of variables. In each iteration, these sets grow progressively, along the edges in this structure. The most that any IN or OUT set can grow is the top abstract value, the set of all variables in the program.



CIS 547 - Software Analysis

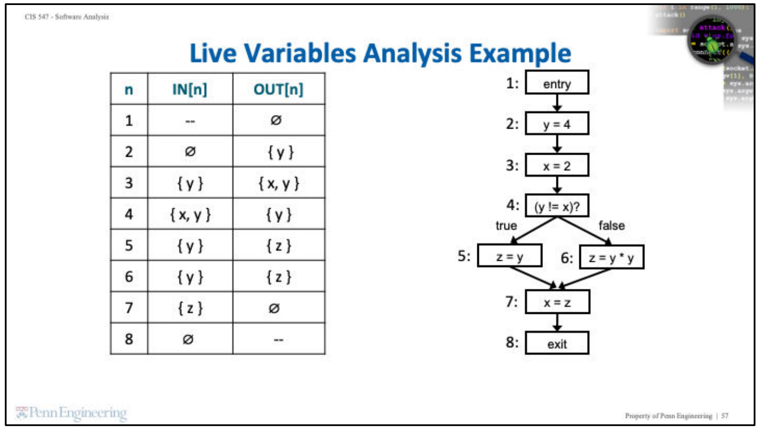
Live Variables Analysis Example

n	IN[n]	OUT[n]
1	--	∅
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	∅
8	∅	--

Penn Engineering

Property of Penn Engineering | 56

Let's work through this program and complete the live variables analysis for it. Remember that the IN set of a node is every live variable before the node, and the OUT set is every variable which is live after the node.



CIS 547 - Software Analysis

Live Variables Analysis Example

n	IN[n]	OUT[n]
1	--	∅
2	∅	{ y }
3	{ y }	{ x, y }
4	{ x, y }	{ y }
5	{ y }	{ z }
6	{ y }	{ z }
7	{ z }	∅
8	∅	--

Penn Engineering

Property of Penn Engineering | 57

We'll start at the exit point. We won't be needing any variables at this point, so it begins with the empty set as its IN set (gesture at IN[8]).

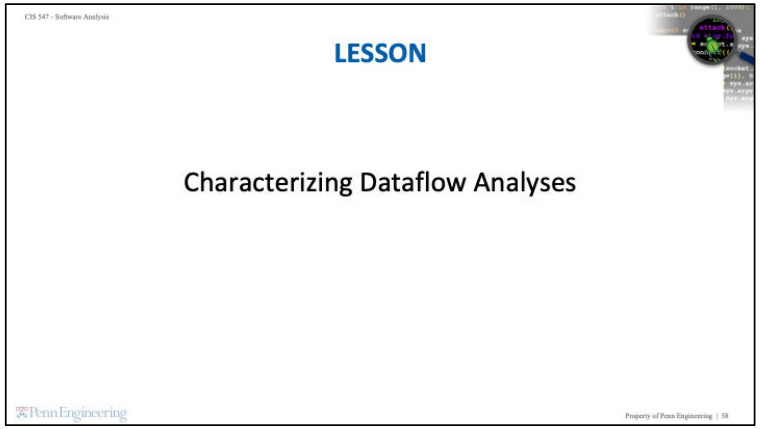
At node 7, the OUT set is the same as the IN set of its successor, node 8 (gesture at OUT[7]). For the IN set, we take whatever our OUT set is, kill any variables which we redefine, and then add any variables which are used in the node. This results in an IN set of z (write {z} in IN[7]).

Both nodes 5 and 6 take the IN set of node 7 as their OUT set (write {z} in OUT[5] and OUT[6]). Node 6 redefines z and uses y, so we remove z from OUT and add in y to obtain the IN set (write {y} in IN[6]). Node 5 also redefines z and uses y, so we kill z and add y to get the IN set (write {y} in IN[5]).

Next we have the condition node 4. Its OUT set is {y}, the union of the IN sets of nodes 5 and 6 (write {y} in OUT[4]). It uses both y and x without redefining any variables, so its IN set is {x,y} (write {x,y} in IN[4]). We then copy over this IN set to the OUT set of node 3 (write {x,y} in OUT[3]).

The rest of the example does not use any variables but sets them to constants. Node 3 kills x in between its OUT and IN sets (write {y} in IN[3] and OUT[2]). Node 2 kills y in between its OUT and IN sets (write ∅ in OUT[1]).

What's interesting is that even though this program has 3 variables x, y, and z, at no point are more than two of these three variables simultaneously live. This information can be used to generate assembly code that uses only two instead of three registers for storing the contents of these variables. Using fewer registers in turn can generate more efficient assembly code, by avoiding the need to store and load the contents of these variables in memory.



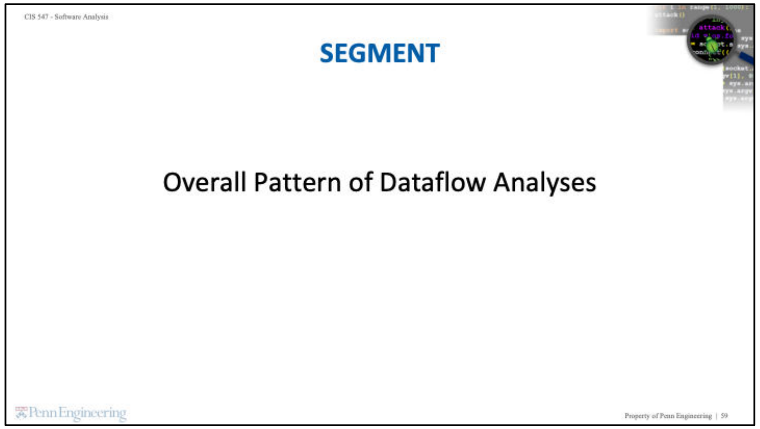
CIS 547 - Software Analysis

LESSON

Characterizing Dataflow Analyses

Penn Engineering

Property of Penn Engineering | 58



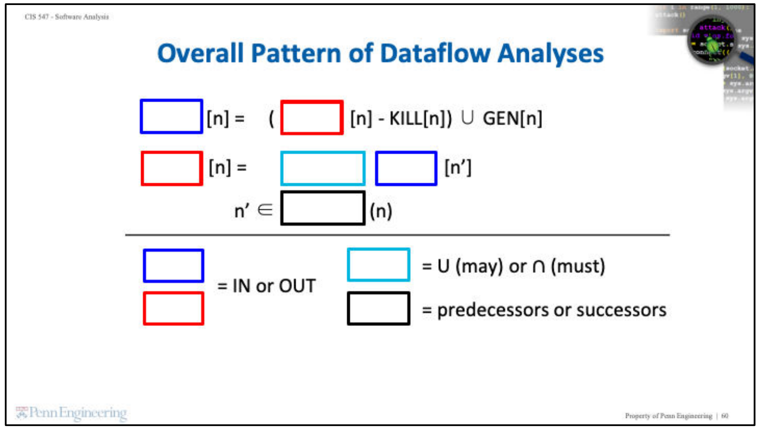
CIS 547 - Software Analysis

SEGMENT

Overall Pattern of Dataflow Analyses

Penn Engineering

Property of Penn Engineering | 59



CIS 547 - Software Analysis

Overall Pattern of Dataflow Analyses

🟦[n] = (🟥[n] - KILL[n]) ∪ GEN[n]

🟥[n] = ⬜ 🟦[n']
n' ∈ ⬛(n)

🟦 🟥 = IN or OUT

⬜ = ∪ (may) or ∩ (must)

⬛ = predecessors or successors

Penn Engineering

Property of Penn Engineering | 60

As you may have noticed at this point, the four dataflow analyses that we discussed follow a common pattern in the two operations that they apply.

This pattern is as follows (show the pattern).

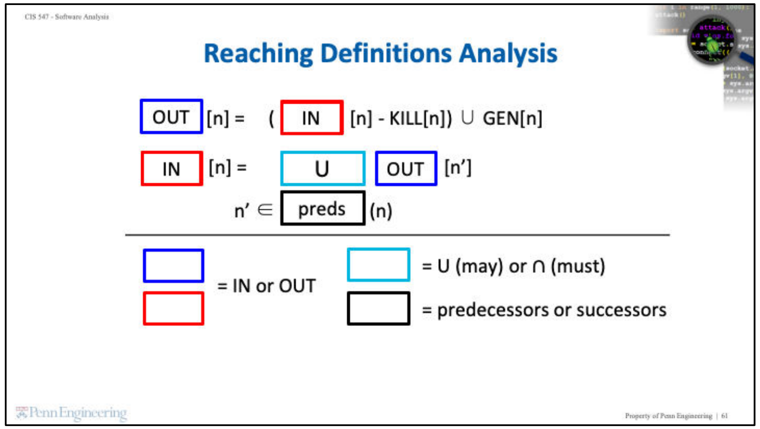
The blue and red boxes represent the IN or OUT sets. The purple box represents the set union or set intersection operator. The black box represents the immediate predecessors or immediate successors of a node.

Each of our four dataflow analyses corresponds to a different instantiation of these boxes. Although this looks like a lot of choices, there are in fact only two: first, whether the analysis propagates information forward or backward, and second, whether the analysis computes “may” or “must” information.

We already saw what forward versus backward propagation looks like when we discussed the four analyses earlier. Forward versus backward propagation is decided by mapping the blue and red boxes to IN and OUT sets appropriately, and by mapping the black box to predecessors or successors accordingly.

Now let’s see what “may” versus “must” information means. Intuitively, an analysis is said to compute “may” facts if those facts hold along some path in the control-flow graph. In contrast, an analysis is said to compute “must” facts if those facts hold along all paths. Thus, the “may” versus “must” property of an analysis is decided by mapping the purple box to the set union operator in the case of "may" analysis and to the set intersection operator in the case of "must" analysis.

Now that we have reviewed the overall pattern, let's instantiate it in turn for each of our four classic dataflow analyses.



Reaching Definitions Analysis

$$\boxed{\text{OUT}}[n] = (\boxed{\text{IN}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{IN}}[n] = \underset{n' \in \boxed{\text{preds}}(n)}{\boxed{\cup}} \boxed{\text{OUT}}[n']$$

□ □ = IN or OUT

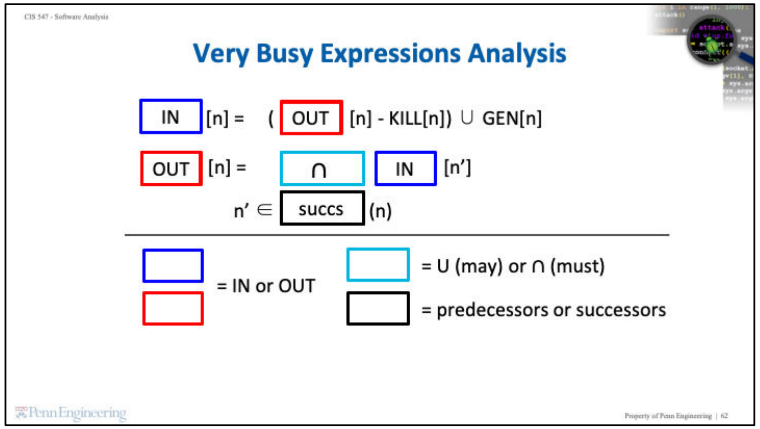
□ = U (may) or ∩ (must)

□ = predecessors or successors

We'll start with reaching-definitions analysis, whose rules we examined earlier in the lesson.

This analysis computes "may" information, which is evident from the goal of the analysis, which is to find definitions that could reach a program point along some path. Therefore, we fill in the purple box with set union.

Furthermore, the analysis propagates this information about reaching definitions forward in the control-flow graph. Hence, we fill in the blue boxes with OUT sets, the red boxes with IN sets, and the black box with predecessors.



Very Busy Expressions Analysis

IN [n] = (OUT [n] - KILL[n]) ∪ GEN[n]

OUT [n] = ∩ IN [n']

n' ∈ succs (n)

□ □ = IN or OUT

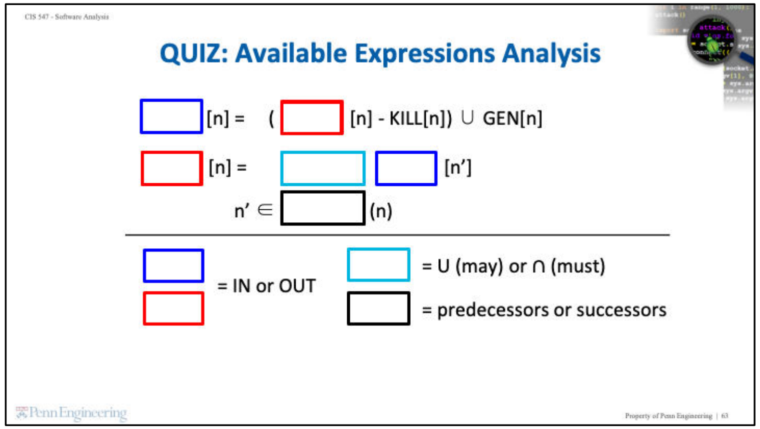
□ = ∪ (may) or ∩ (must)

□ = predecessors or successors

Next let's look at very-busy-expressions analysis. This is the polar opposite of reaching definitions analysis: it is a backward, "must" analysis.

This analysis computes "must" information, because the goal of the analysis is to find expressions that are used along all paths before any variable occurring in them is redefined. Therefore, we fill in the purple box with the set intersection operator.

Furthermore, the analysis propagates this information about very busy expressions backwards in the control-flow graph. Therefore, we fill in the blue boxes with IN sets, the red boxes with OUT sets, and the black box with successors.



QUIZ: Available Expressions Analysis

□ [n] = (□ [n] - KILL[n]) ∪ GEN[n]

□ [n] = □ □ [n']

n' ∈ □ (n)

□ □ = IN or OUT

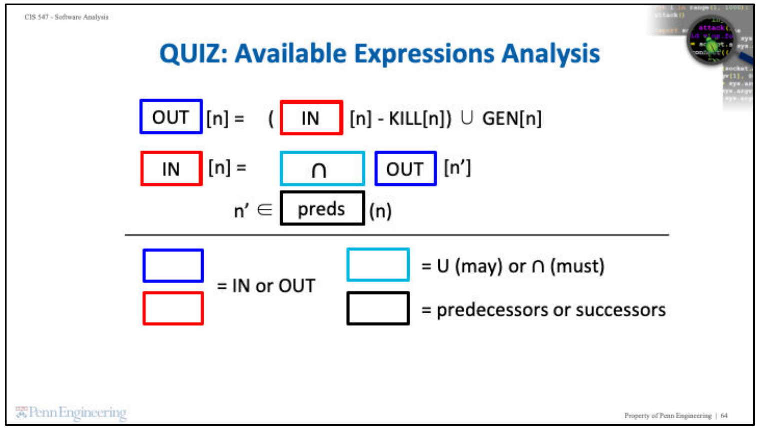
□ = ∪ (may) or ∩ (must)

□ = predecessors or successors

{QUIZ SLIDE}

Now let's fill in the pattern for available expressions analysis. We will do this in the form of an exercise.

Fill in the six boxes with the appropriate values. Type either the word "union" or "intersect" in the purple box, type either predecessors or successors in the black box, and type either "IN" or "OUT" in the red and blue boxes. Remember to use the same value in the two blue boxes, and likewise, the same value in the two red boxes.



CIS 547 - Software Analysis

QUIZ: Available Expressions Analysis

OUT [n] = (IN [n] - KILL[n]) ∪ GEN[n]

IN [n] = ∩ OUT [n']

n' ∈ preds (n)

□ □ = IN or OUT

□ = U (may) or ∩ (must)

□ = predecessors or successors

Penn Engineering

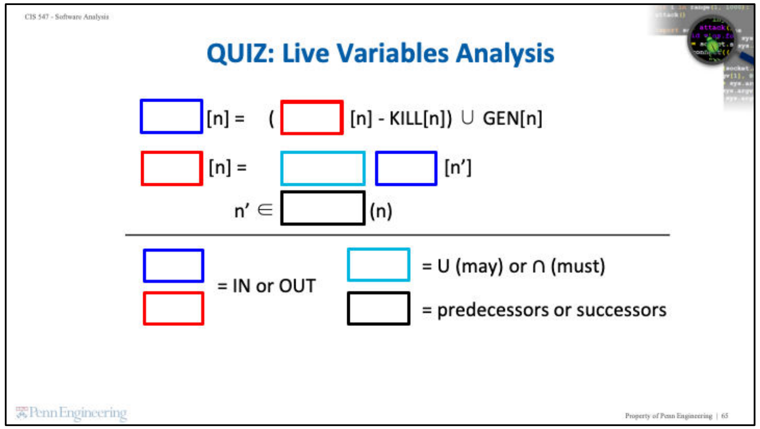
Property of Penn Engineering | 64

{SOLUTION SLIDE}

Let's review the solution. Available-expressions analysis is a forward, must analysis.

It is a must analysis because it seeks to find expressions that are available at a program point, that is, expressions that must have been computed along all paths leading to a point. So we instantiate the purple box with the set intersection operator.

This analysis propagates information about available expressions forward in the control-flow graph. Hence, we instantiate the blue boxes with OUT sets, the red boxes with IN sets, and the black box with predecessors.



CIS 547 - Software Analysis

QUIZ: Live Variables Analysis

□ [n] = (□ [n] - KILL[n]) ∪ GEN[n]

□ [n] = □ □ [n']

n' ∈ □ (n)

□ □ = IN or OUT

□ = U (may) or ∩ (must)

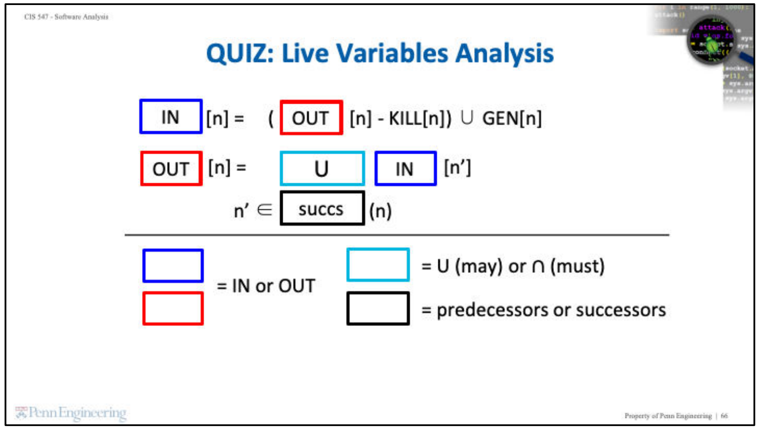
□ = predecessors or successors

Penn Engineering

Property of Penn Engineering | 65

{QUIZ SLIDE}

Finally, let's instantiate the pattern for live variables analysis. Let's do this in the form of an exercise as well. Fill in the six boxes with the appropriate values.



CIS 547 - Software Analysis

QUIZ: Live Variables Analysis

$$\boxed{\text{IN}}[n] = (\boxed{\text{OUT}}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\boxed{\text{OUT}}[n] = \underset{n' \in \boxed{\text{succs}}(n)}{\boxed{\cup}} \boxed{\text{IN}}[n']$$

□ □ = IN or OUT

□ = U (may) or ∩ (must)

□ = predecessors or successors

Penn Engineering

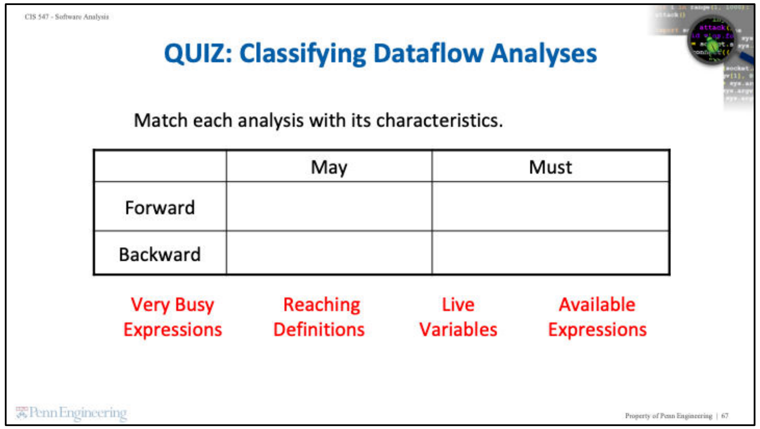
Property of Penn Engineering | 66

{SOLUTION SLIDE}

Let's review the solution. Live-variables analysis is a backward, may analysis.

It is a may analysis because it seeks to find live variables: variables that may be used along some path before being re-defined. So we fill in the purple box with the set union operator.

This analysis propagates information about live variables backwards in the control-flow graph. Hence, we instantiate the blue boxes with IN sets, the red boxes with OUT sets, and the black box with successors.



CIS 547 - Software Analysis

QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

	May	Must
Forward		
Backward		

Very Busy Expressions

Reaching Definitions

Live Variables

Available Expressions

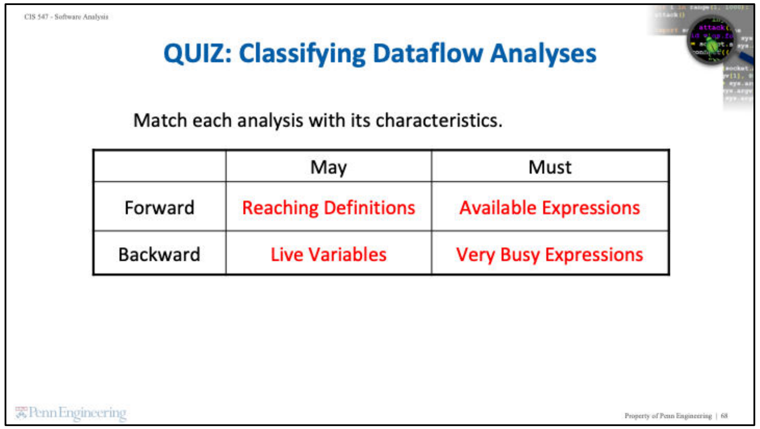
Penn Engineering

Property of Penn Engineering | 67

{QUIZ SLIDE}

We have seen four different dataflow analyses along with their characteristics along two important dimensions: forward versus backward and may versus must.

Let's wrap up the discussion with a brief review of these characteristics. Match each of the four dataflow analyses with their corresponding properties. Type in the letter of the corresponding analysis into each box.



CIS 547 - Software Analysis

QUIZ: Classifying Dataflow Analyses

Match each analysis with its characteristics.

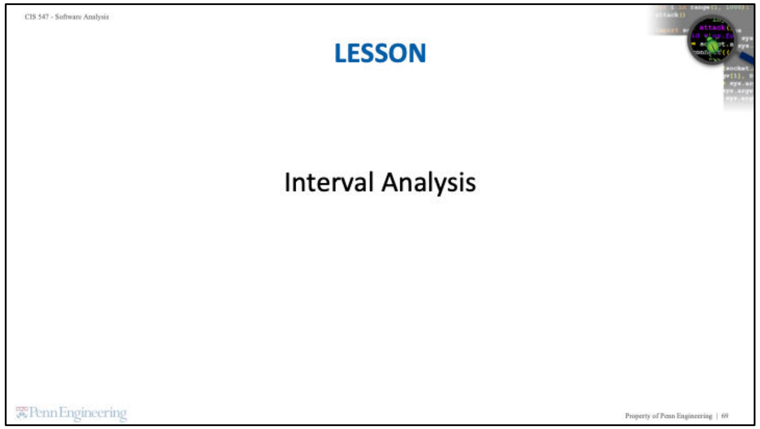
	May	Must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Property of Penn Engineering | 68

{SOLUTION SLIDE}

Let’s review the solution.

Reaching definitions analysis is a forward, may analysis. Available expressions analysis is a forward, must analysis. Live variables analysis is a backward, may analysis. And finally, very busy expressions analysis is a backward, must analysis.

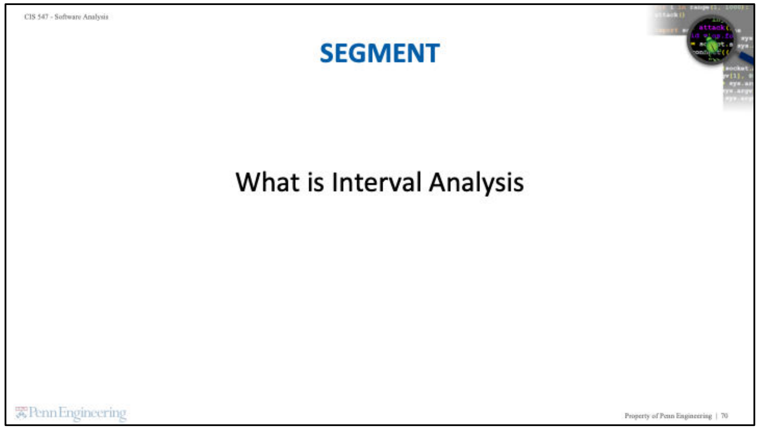


CIS 547 - Software Analysis

LESSON

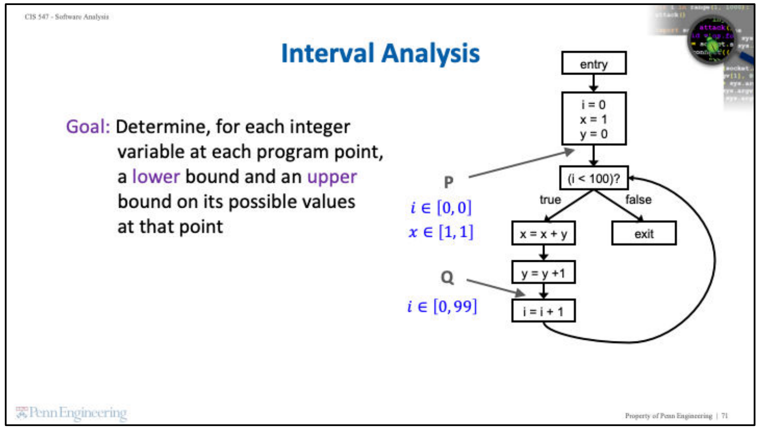
Interval Analysis

Property of Penn Engineering | 69



SEGMENT

What is Interval Analysis



Interval Analysis

Goal: Determine, for each integer variable at each program point, a lower bound and an upper bound on its possible values at that point

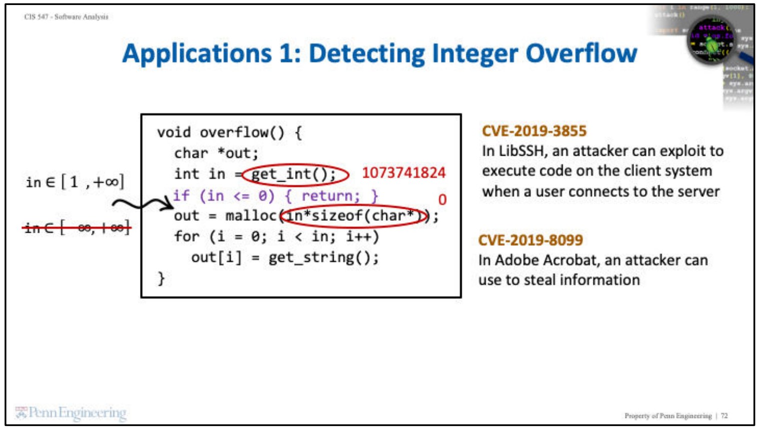
Recall that besides the classic dataflow analyses from the literature on compilers, dataflow analyses also underlie a variety of bug-finding tools that are used in the software industry today. We will delve into one of these analyses next: Interval Analysis.

The goal of Interval Analysis is to compute, for each integer variable at each program point, a lower bound and an upper bound on its possible values at that point.

For instance, consider the program point P.

At this point, variable i's lower and upper bounds are both 0. Similarly, variable x's lower and upper bounds are both 1.

On the other hand, consider program point Q. Since this point is inside a loop, variable i's lower bound is 0, from the first iteration of the loop, and its upper bound is 99, from the last iteration of the loop.



CIS 547 - Software Analysis

Applications 1: Detecting Integer Overflow

in ∈ [1 ,+∞]

~~in ∈ [−∞, +∞]~~

```
void overflow() {
  char *out;
  int in = get_int();        1073741824
  if (in <= 0) { return; }            0
  out = malloc(in*sizeof(char*));
  for (i = 0; i < in; i++)
    out[i] = get_string();
}
```

CVE-2019-3855
In LibSSH, an attacker can exploit to execute code on the client system when a user connects to the server

CVE-2019-8099
In Adobe Acrobat, an attacker can use to steal information

Penn Engineering

Property of Penn Engineering | 72

Interval analysis has many applications in finding security bugs. In fact, around a third of the serious bugs in the Linux kernel are integer-related errors.

One of these kinds of bugs is integer overflows.

This kind of bug is caused by the fact that in C, integer variables have a maximum possible value dictated by the underlying hardware architecture, and when that value is exceeded, the result wraps around.

Let’s look at an example: this C code fragment starts by reading an integer input from the user through the get_int() function.

Assume that the user provides a very large number such as this 10-digit number.

Then the program calls the malloc() function to allocate memory. Suppose the size of pointer to char in our architecture is 4.

As a result, the argument to malloc(), which multiples the very large number provided by the user with 4, results in an overflow and wraps around to zero.

Almost all malloc implementations allocate a buffer with size zero without complaining.

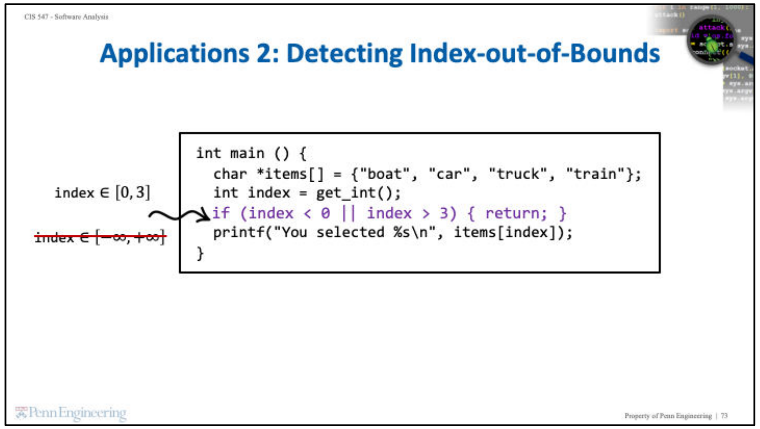
Consequently, the write to the allocated buffer overruns the space allotted to the buffer, which is 0 bytes.

This kind of bug is a security vulnerability that can be exploited by attackers. Two recent examples include integer overflows in LibSSH and Adobe Acrobat that can lead to serious exploits such as executing arbitrary code and leaking sensitive information.

Running interval analysis on this program can reveal that the argument of malloc could be zero. Specifically, the analysis will infer that, just before the call to malloc, the variable ‘in’ can range from minus infinity to plus infinity.

This bug can be fixed by adding a check before the call to malloc. In the fixed program, interval analysis infers that the variable ‘in’ is strictly positive just before the call to malloc.

Note, however, that while this fix avoids the common problem of invoking malloc with argument zero, there is still a possibility of an integer overflow. Possible solutions include strengthening the check, for instance, checking that the value of variable ’in’ is less than or equal to the value of the expression ‘in’ times the size of pointer to char, or using calloc instead of malloc – calloc uses separate arguments for the number of elements to be allocated, in this case the value of variable ‘in’, and the size of each element, in this case the size of pointer to char.



CIS 547 - Software Analysis

Applications 2: Detecting Index-out-of-Bounds

index ∈ [0, 3]

~~index ∈ [−∞, +∞]~~

```
int main () {
  char *items[] = {"boat", "car", "truck", "train"};
  int index = get_int();
  if (index < 0 || index > 3) { return; }
  printf("You selected %s\n", items[index]);
}
```

Penn Engineering

Property of Penn Engineering | 73

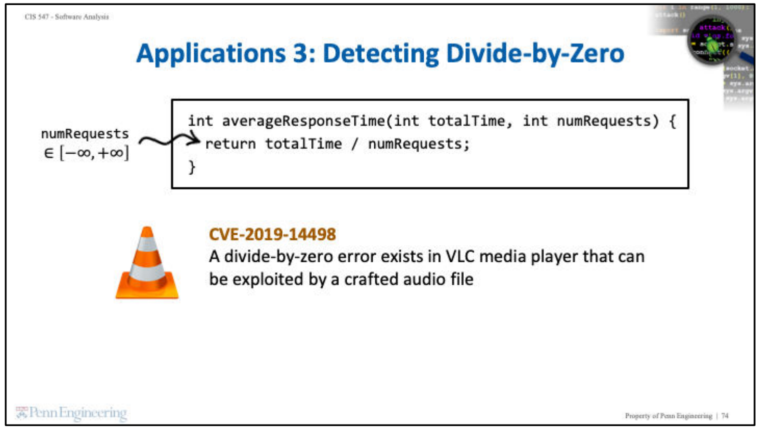
Another common bug that is detectable by interval analysis is array index-out-of-bounds. In C, nothing prevents one from accessing beyond the bounds of an array, so the programmer must check for it explicitly.

If the programmer neglects to do so, the program results in undefined behavior from accessing memory past the end of the array. This undefined behavior in turn can be exploited by an attacker for various ends, such as to gain unauthorized access.

Interval analysis can help detect the bug in this example program by warning the programmer that the set of possible inputs for variable ‘index’ is from “-infinity” to “+infinity”.

Once the bug is found, the fix is straightforward: simply introduce explicit bounds checking before accessing the array.

In the fixed program, interval analysis infers that the variable ‘index’ is in the range 0 to 3 just before indexing into the ‘items’ array.



CIS 547 - Software Analysis

Applications 3: Detecting Divide-by-Zero

numRequests ∈ [−∞, +∞]

```
int averageResponseTime(int totalTime, int numRequests) {
 return totalTime / numRequests;
}
```

CVE-2019-14498
A divide-by-zero error exists in VLC media player that can be exploited by a crafted audio file

Penn Engineering

Property of Penn Engineering | 74

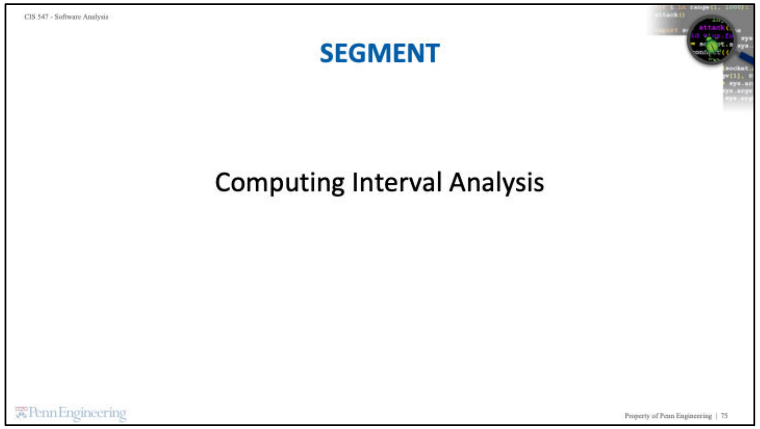
Finally, another kind of bug that interval analysis can detect is divide-by-zero bugs.

Division by zero is not *inherently* a security vulnerability. However, it can cause an application server to crash and stay offline by performing a division by zero and cause a denial of service (or DoS) attack.

For instance, consider the following function used for computing the average response time in the implementation of a network protocol. This function contains a division in which the denominator is not checked for being non-zero.

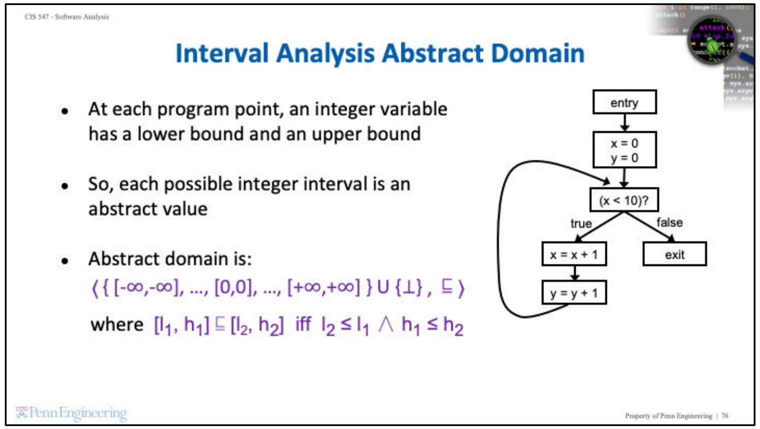
An interval analysis reports that numRequests could be zero by virtue of the fact that its range of possible values is from minus infinity to plus infinity.

As another example, here is a security vulnerability discovered in a well-known media player, VLC, which was caused by a divide-by-zero error that can be triggered by playing a carefully crafted audio file.



SEGMENT

Computing Interval Analysis



Interval Analysis Abstract Domain

- At each program point, an integer variable has a lower bound and an upper bound
- So, each possible integer interval is an abstract value
- Abstract domain is:
  $\langle \{ [-\infty,-\infty], \ldots, [0,0], \ldots, [+\infty,+\infty] \} \cup \{\bot\}, \sqsubseteq \rangle$
  where $[l_1, h_1] \sqsubseteq [l_2, h_2]$ iff $l_2 \leq l_1 \wedge h_1 \leq h_2$

Now that the importance of interval analysis is clear, let's dive into how it works.

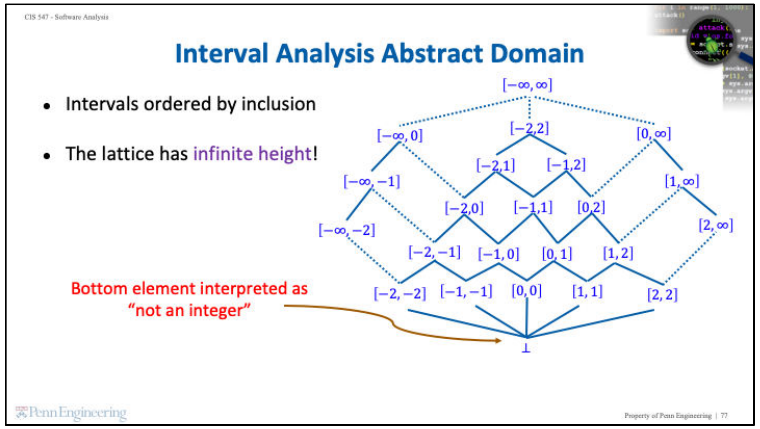
Let's begin with the abstract domain for interval analysis.

At each program point, an integer variable has a lower bound and an upper bound.

So, each possible integer interval is an abstract value.

So, the abstract domain is the set of all possible integer intervals. Additionally, it includes a bottom element, needed for the structure to be a lattice as we shall see shortly.

The interval values are partially ordered by inclusion: an interval [l1, h1] is at least as precise as interval [l2, h2] if and only if the former is properly contained in the latter, that is, l1 >= l2 and h1 <= h2.



CIS 547 - Software Analysis

Interval Analysis Abstract Domain

- Intervals ordered by inclusion
- The lattice has infinite height!

Bottom element interpreted as “not an integer”

Penn Engineering

Property of Penn Engineering | 77

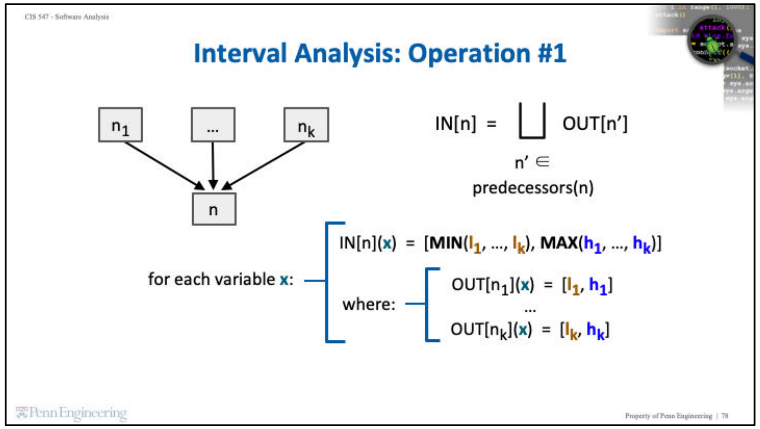
Here is a visualization of the abstract domain for interval analysis.

Notice that the intervals are ordered by inclusion, for instance, there is an edge from interval [0,0] to interval [0,1] since the former is included in the latter.

The bottom element is interpreted as ``not an integer”. It serves as the abstract value of the result of division-by-zero or the abstract value for integer variables at program points that are unreachable.

You might notice a fundamental difference between this structure and the structure of the abstract domains for the dataflow analyses we discussed previously. This one has infinite height.

The infinite height poses a problem: the fixed-point computation approach that we have seen so far requires this structure to have finite height in order to guarantee termination. Shortly, we will see how this problem can be circumvented using a technique called widening. But before that, let’s first learn about the operations of interval analysis.



CIS 547 - Software Analysis

Interval Analysis: Operation #1

$$\mathrm{IN}[n] = \bigsqcup_{n' \in \mathrm{predecessors}(n)} \mathrm{OUT}[n']$$

for each variable **x**:

$$\mathrm{IN}[n](x) = [\mathbf{MIN}(l_1, \ldots, l_k), \mathbf{MAX}(h_1, \ldots, h_k)]$$

where:

$$\mathrm{OUT}[n_1](x) = [l_1, h_1]$$

...

$$\mathrm{OUT}[n_k](x) = [l_k, h_k]$$

Penn Engineering

Property of Penn Engineering | 78

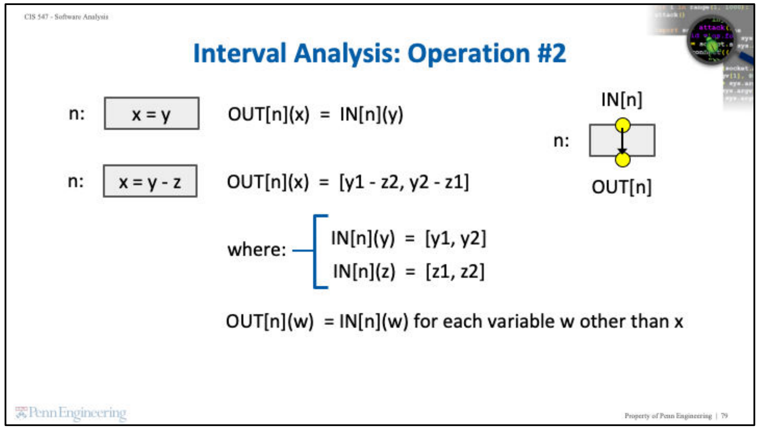
Here’s the first of the two operations of interval analysis.

This operation computes the analysis result at the entry of a particular node n in the control-flow graph from the analysis result at the exit of that node’s immediate predecessors n1 thru nk.

The operation computes this by merging the intervals of each variable.

In order to merge multiple intervals that correspond to a variable, we compute the minimum of the lower bounds to be the new lower bound, and the maximum of the upper bounds to be the new upper bound.

Let us review this for a variable x. Suppose its lower and upper bounds computed by the analysis thus far at the exit of node n1 are l1 and h1 respectively. And similarly, at the exit of node nk are lk and hk respectively. Then, its lower bound at the entry of node n is the minimum of l1 thru lk, and its upper bound is the maximum of h1 thru hk.



CIS 547 - Software Analysis

Interval Analysis: Operation #2

n: x = y   OUT[n](x) = IN[n](y)

n: x = y - z   OUT[n](x) = [y1 - z2, y2 - z1]

where: IN[n](y) = [y1, y2]
IN[n](z) = [z1, z2]

OUT[n](w) = IN[n](w) for each variable w other than x

PennEngineering

Property of Penn Engineering | 79

The second operation of interval analysis computes the analysis result at the exit of a particular node from the analysis result at the entry of that node. The details of this operation depend on the kind of statement that occurs at the node we are looking at.

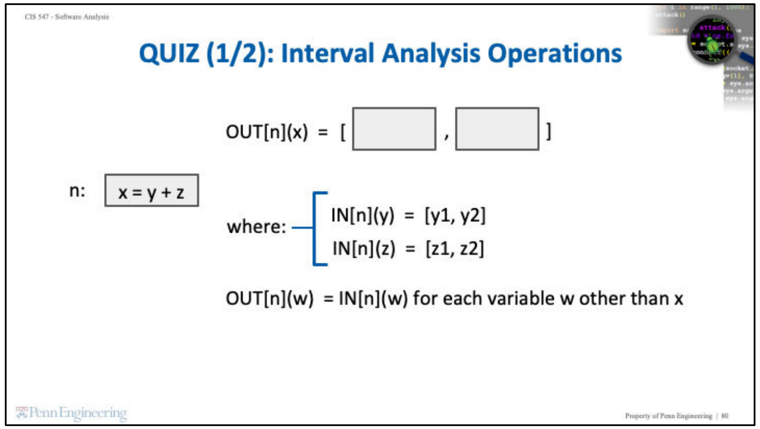
Let's look at the kinds of possible statements in the WHILE language.

Suppose the statement at node n is an assignment statement in which y is assigned to x. Since y is copied to x, we set the interval of x at the exit of node n to be the interval of y at the entry of node n.

Next, let's look at the subtraction statement in the WHILE language.

If the statement at node n is a subtraction statement in which y is subtracted from z, and the result is saved in x, then the interval of x at the exit of node n depends on the intervals of y and z at the entry of node n. Suppose those intervals are y1..y2 and z1..z2, respectively. Then, the lower bound of x is set to the lower bound of y minus the upper bound of z, that is y1 – z2, and the upper bound of x is set to the upper bound of y minus the lower bound of z, or y2 – z1.

Lastly, for each of these two kinds of statements, the interval of each variable w other than x at the exit of node n remains the same as the interval of that variable at the entry of node n, since neither of these statements modify any variable other than x.



CIS 547 - Software Analysis

QUIZ (1/2): Interval Analysis Operations

OUT[n](x) = [☐ , ☐]

n: x = y + z

where: IN[n](y) = [y1, y2]
IN[n](z) = [z1, z2]

OUT[n](w) = IN[n](w) for each variable w other than x

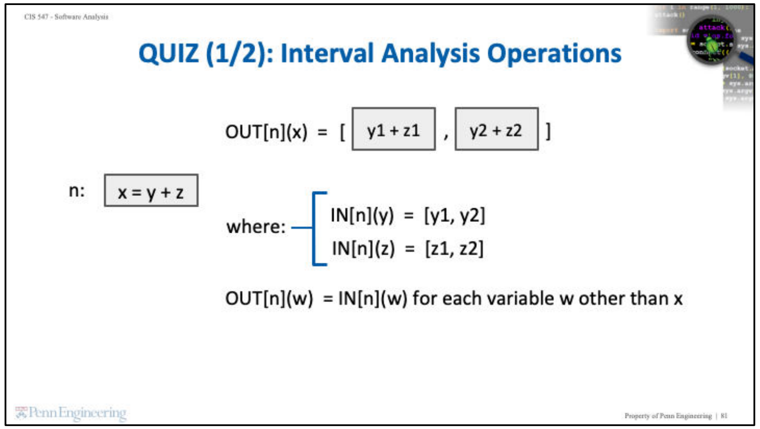
PennEngineering

Property of Penn Engineering | 80

{QUIZ SLIDE}

To check your understanding of how interval analysis operates, let us do a quiz in two parts, for two more arithmetic operations.

Suppose the statement at node n adds y and z, and stores the result in x. You are given the lower and upper bounds of variables y and z at the entry of node n: they are y1 and y2 respectively for variable y, and z1 and z2 respectively for variable z. Write the lower and upper bounds of variable x at the exit of node n.



CIS 547 - Software Analysis

QUIZ (1/2): Interval Analysis Operations

OUT[n](x) = [y1 + z1 , y2 + z2]

n: x = y + z

where: IN[n](y) = [y1, y2]
IN[n](z) = [z1, z2]

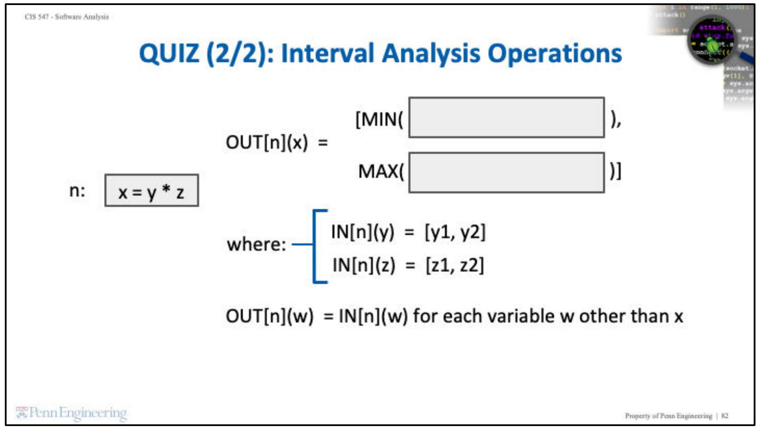
OUT[n](w) = IN[n](w) for each variable w other than x

Penn Engineering

Property of Penn Engineering | 81

{SOLUTION SLIDE}

The add operation for interval analysis is relatively straightforward: the lower bound of x is simply the sum of the lower bounds of y and z, that is, y1 + z1. Similarly, the upper bound of x is simply the sum of the upper bounds of y and z, that is, y2 + z2.



CIS 547 - Software Analysis

QUIZ (2/2): Interval Analysis Operations

OUT[n](x) = [MIN( ), MAX( )]

n: x = y * z

where: IN[n](y) = [y1, y2]
IN[n](z) = [z1, z2]

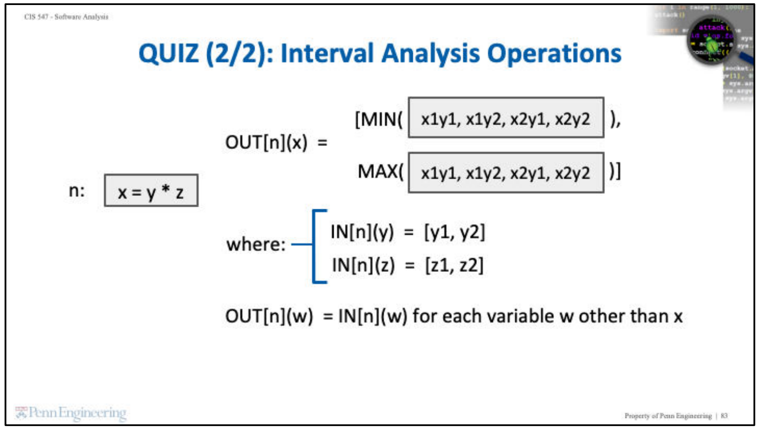
OUT[n](w) = IN[n](w) for each variable w other than x

Penn Engineering

Property of Penn Engineering | 82

{QUIZ SLIDE}

In the second part of this quiz, let us look at a more complex arithmetic operation: integer multiplication. Write the lower and upper bounds of variable x at the exit of node n, which assigns the result of multiplying y and z to x. You are given the lower and upper bounds of variables y and z at the entry of node n, that is, y1 and y2 respectively for variable y, and z1 and z2 respectively for variable z. As a hint, the lower bound of x will be the minimum of certain expressions over y1, y2, z1, and z2. Similarly, the upper bound of x will be the maximum of certain expressions over y1, y2, z1, and z2.



CIS 547 - Software Analysis

QUIZ (2/2): Interval Analysis Operations

n: x = y * z

OUT[n](x) = [MIN(x1y1, x1y2, x2y1, x2y2), MAX(x1y1, x1y2, x2y1, x2y2)]

where:
- IN[n](y) = [y1, y2]
- IN[n](z) = [z1, z2]

OUT[n](w) = IN[n](w) for each variable w other than x

Penn Engineering

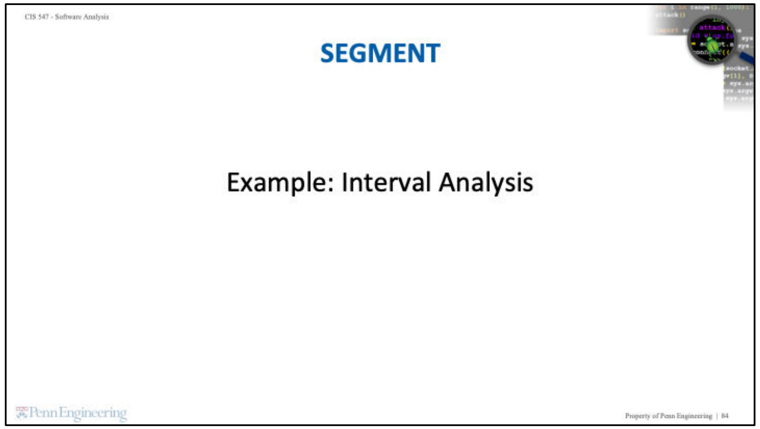
Property of Penn Engineering | 83

{SOLUTION SLIDE}

Let’s review the solution. First, all four combinations of multiplying the lower bounds and the upper bounds of y and z are computed. Then, the minimum of these values is the lower bound of x, while the maximum of these values is its upper bound.

You can read an article on interval arithmetic linked below to understand the intuition behind this calculation. The article also shows the interval analysis operations for other kinds of statements such as integer division.

[https://en.wikipedia.org/wiki/Interval_arithmetic]



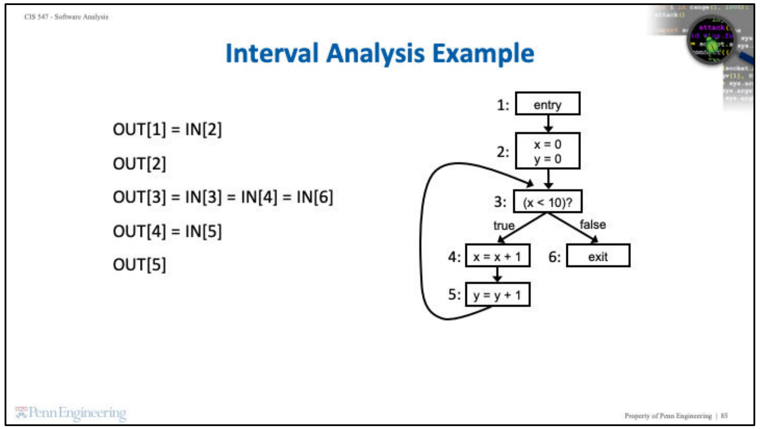
CIS 547 - Software Analysis

SEGMENT

Example: Interval Analysis

Penn Engineering

Property of Penn Engineering | 84



CIS 547 - Software Analysis

Interval Analysis Example

OUT[1] = IN[2]

OUT[2]

OUT[3] = IN[3] = IN[4] = IN[6]

OUT[4] = IN[5]

OUT[5]

Penn Engineering

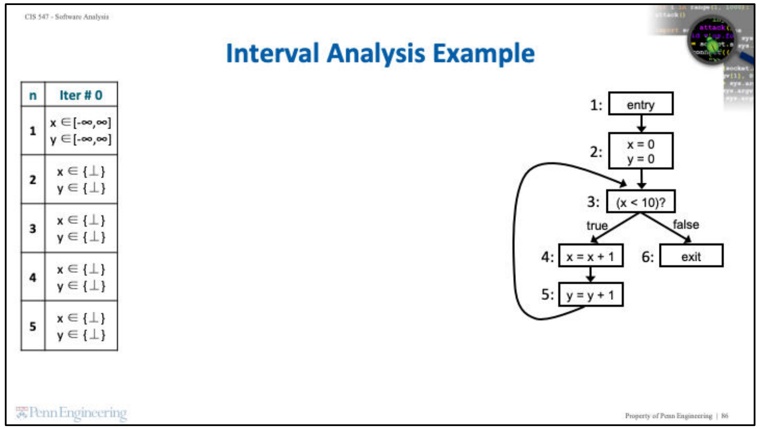
Property of Penn Engineering | 85

Now, let us walk through a simple example to illustrate how interval analysis operates step by step.

In this example, to avoid clutter, we will not write all the IN and OUT results, since some of them are the same.

For instance, the result at the exit of node 1 is the same as the result at the entry of node 2. Likewise, the result at the exit of node 4 is the same as the result at the entry of node 5.

So we will only show the results at the exit of nodes 1, 2, 3, 4, and 5.



CIS 547 - Software Analysis

Interval Analysis Example

n	Iter # 0
1	x ∈[-∞,∞] y ∈[-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}

3	x ∈ {⊥}
	y ∈ {⊥}

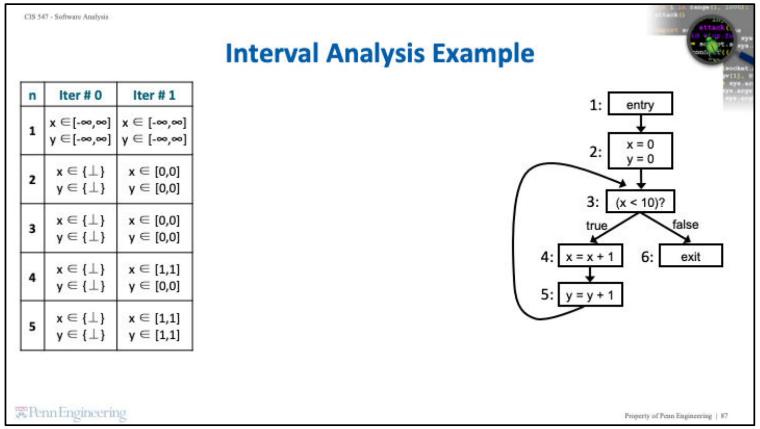
4	x ∈ {⊥}
	y ∈ {⊥}

5	x ∈ {⊥}
	y ∈ {⊥}

Penn Engineering

Property of Penn Engineering | 86

This program has two integer variables x and y. The analysis initializes the result at the exit of each node n by setting the abstract value of each of these variables to bottom, except at the exit of the entry node, where we set the abstract value of each of the variables to top, or [-infty, +infty]. The analysis then performs the chaotic iteration process, applying the operations we just discussed to update the current analysis result at the entry and exit of each node, until the result stops changing.



CIS 547 - Software Analysis

Interval Analysis Example

n	Iter # 0	Iter # 1
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]

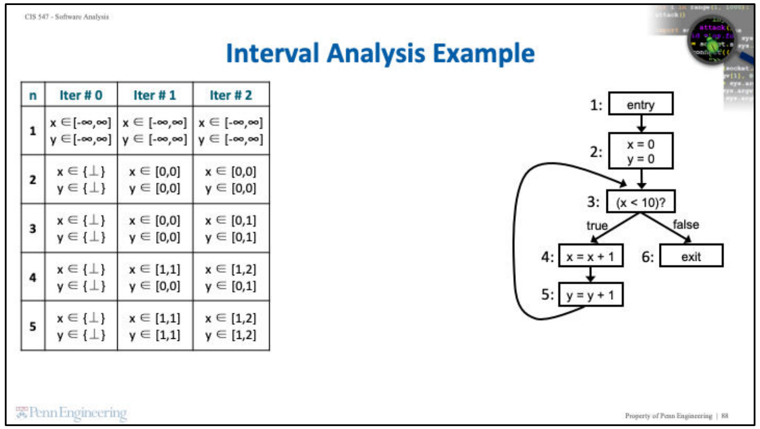
3	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]

4	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [0,0]

5	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [1,1]

Property of Penn Engineering | 87

After the first iteration, the abstract values of variables x and y at the exit of each of nodes 1 thru 5 are as follows. For instance, at the exit of node 1, the abstract value of x and y is still top, capturing the fact that both are uninitialized and therefore could have arbitrary initial values in an execution. On the other hand, at the exit of node 4, the abstract value of x is 1 but the abstract value of y is 0, as expected.



CIS 547 - Software Analysis

Interval Analysis Example

n	Iter # 0	Iter # 1	Iter # 2
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,0]
			y ∈ [0,0]

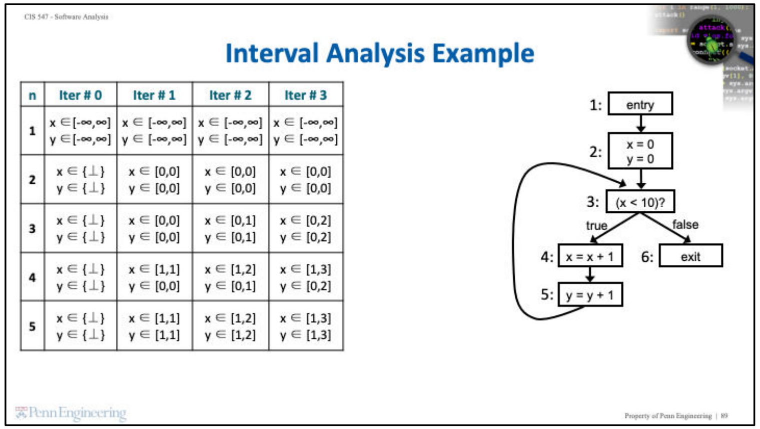
3	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,1]
			y ∈ [0,1]

4	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [0,0]
			x ∈ [1,2]
			y ∈ [0,1]

5	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [1,1]
			x ∈ [1,2]
			y ∈ [1,2]

Property of Penn Engineering | 88

In the next iteration, the result at the exit of nodes 3, 4, and 5 is updated, reflecting the fact that these nodes are part of a loop. Intuitively, the analysis has deduced the ranges of variables x and y in upto two iterations of the loop. Since the result changed in this iteration, the analysis makes another pass.



CIS 547 - Software Analysis

Interval Analysis Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,0]
			y ∈ [0,0]
				x ∈ [0,0]
				y ∈ [0,0]

3	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,1]
			y ∈ [0,1]
				x ∈ [0,2]
				y ∈ [0,2]

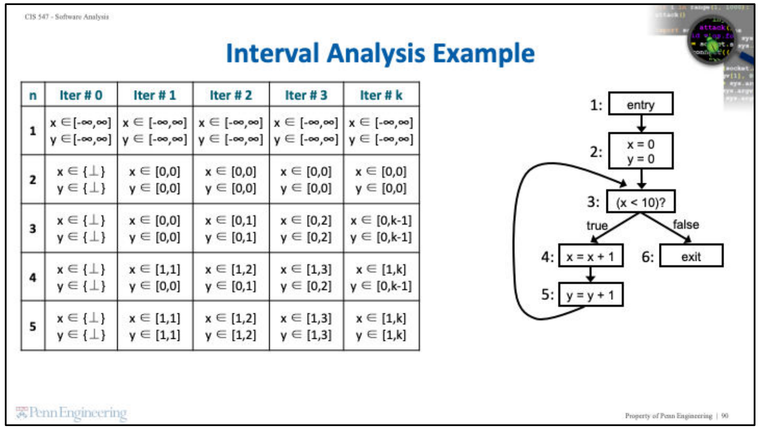
4	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [0,0]
			x ∈ [1,2]
			y ∈ [0,1]
				x ∈ [1,3]
				y ∈ [0,2]

5	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [1,1]
			x ∈ [1,2]
			y ∈ [1,2]
				x ∈ [1,3]
				y ∈ [1,3]

PennEngineering

Property of Penn Engineering | 89

The result at the exit of nodes 3, 4, and 5 is updated further, this time capturing the result of analyzing upto three iterations of the loop. The analysis repeats the process yet again.



CIS 547 - Software Analysis

Interval Analysis Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,0]
			y ∈ [0,0]
				x ∈ [0,0]
				y ∈ [0,0]
					x ∈ [0,0]
					y ∈ [0,0]

3	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,1]
			y ∈ [0,1]
				x ∈ [0,2]
				y ∈ [0,2]
					x ∈ [0,k-1]
					y ∈ [0,k-1]

4	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [0,0]
			x ∈ [1,2]
			y ∈ [0,1]
				x ∈ [1,3]
				y ∈ [0,2]
					x ∈ [1,k]
					y ∈ [0,k-1]

5	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [1,1]
			x ∈ [1,2]
			y ∈ [1,2]
				x ∈ [1,3]
				y ∈ [1,3]
					x ∈ [1,k]
					y ∈ [1,k]

PennEngineering

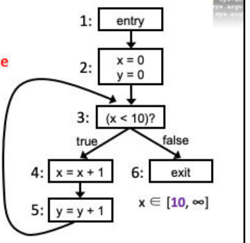
Property of Penn Engineering | 90

Let’s envision the result of the analysis after k iterations.

Interval Analysis Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,9]$ $y \in [0,9]$
4	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,10]$ $y \in [0,9]$
5	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,10]$ $y \in [1,10]$

more precise analysis

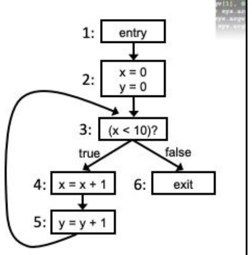


At this point, note that a more sophisticated set of rules for interval analysis, for example, one that distinguishes between true and false branches if the branch condition is simply a comparison between a variable and a constant integer, could produce more precise bounds.

An example of such a condition is "x < 10" at node 3. Therefore, the upper bound of x immediately before node 4 would be 9, and immediately before node 6, the lower bound would be 10. In this case, the analysis would terminate after 11 iterations, analyzing each of the 10 iterations of the loop in the corresponding analysis iteration, and doing a final iteration to ensure that the analysis result has saturated. However, here we are only discussing an interval analysis that does not distinguish between branches, instead abstracting all control-flow conditions with non-deterministic choice. So the analysis continues.

Interval Analysis Example

n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k
1	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$	$x \in [-\infty,\infty]$ $y \in [-\infty,\infty]$
2	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,0]$ $y \in [0,0]$
3	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [0,0]$ $y \in [0,0]$	$x \in [0,1]$ $y \in [0,1]$	$x \in [0,2]$ $y \in [0,2]$	$x \in [0,k-1]$ $y \in [0,k-1]$
4	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [0,0]$	$x \in [1,2]$ $y \in [0,1]$	$x \in [1,3]$ $y \in [0,2]$	$x \in [1,k]$ $y \in [0,k-1]$
5	$x \in \{\perp\}$ $y \in \{\perp\}$	$x \in [1,1]$ $y \in [1,1]$	$x \in [1,2]$ $y \in [1,2]$	$x \in [1,3]$ $y \in [1,3]$	$x \in [1,k]$ $y \in [1,k]$



This was the K-th iteration of the analysis …

Interval Analysis Example

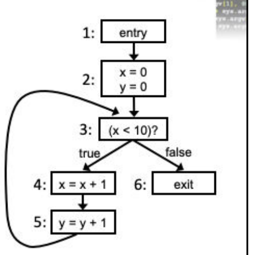
n	Iter # 0	Iter # 1	Iter # 2	Iter # 3	Iter # k	Iter # ∞
1	x ∈[-∞,∞] y ∈[-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]	x ∈ [-∞,∞] y ∈ [-∞,∞]

2	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,0]
			y ∈ [0,0]
				x ∈ [0,0]
				y ∈ [0,0]
					x ∈ [0,0]
					y ∈ [0,0]
						x ∈ [0,0]
						y ∈ [0,0]

3	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [0,0]
		y ∈ [0,0]
			x ∈ [0,1]
			y ∈ [0,1]
				x ∈ [0,2]
				y ∈ [0,2]
					x ∈ [0,k-1]
					y ∈ [0,k-1]
						x ∈ [0,∞]
						y ∈ [0,∞]

4	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [0,0]
			x ∈ [1,2]
			y ∈ [0,1]
				x ∈ [1,3]
				y ∈ [0,2]
					x ∈ [1,k-1]
					y ∈ [0,k-1]
						x ∈ [1,∞]
						y ∈ [0,∞]

5	x ∈ {⊥}
	y ∈ {⊥}
		x ∈ [1,1]
		y ∈ [1,1]
			x ∈ [1,2]
			y ∈ [1,2]
				x ∈ [1,3]
				y ∈ [1,3]
					x ∈ [1,k-1]
					y ∈ [1,k-1]
						x ∈ [1,∞]
						y ∈ [1,∞]



If we repeat the pattern infinitely many times, we would get the final analysis result in the infinite-th iteration. However, this does not make sense in practice. The issue is that the conventional fixed-point computation requires an infinite number of iterations to converge.

The solution to overcome this problem is a technique called "widening".

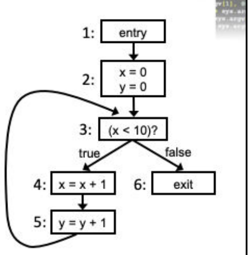
Widening

- In infinite-height lattice, the fixed-point computation does not terminate!
- Solution: Widening

  Infinite ascending chain:

  y → ⊥ , [1,1], [1,2], [1,3], …, [1,k-1], …, [1,∞]

  Finite ascending chain: y → ⊥ , [1,1], [1,2], [1,∞]



As we saw in the interval analysis example, there are times when it is useful to define an abstract domain as a lattice of infinite height. We would like a mechanism for ensuring that the analysis will terminate even for such an abstract domain. One way to do this is to find situations where the value may be ascending infinitely at a given program point, and effectively shorten the chain to a finite height.

We can do so with a widening operator.

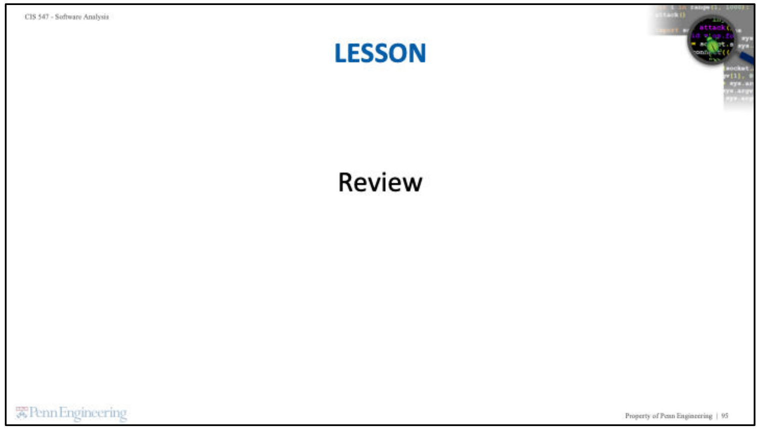
Recall the previous example. At the exit of node 5, the abstract values of y across the analysis iterations were bottom, [1,1], [1,2], [1,3], etc. In other words, they form an ascending chain. Which means that in every iteration, the computed range is a higher element in the lattice.

A widening operator's purpose is to compress infinite chains to finite length. The widening operator considers the most recent two elements in a chain. If the second is higher than the first, the widening operator can choose to jump up in the lattice, potentially skipping elements in the chain.

For example, one way to cut the ascending chain above down to a finite height is to observe that the upper limit for y is increasing, and therefore assume the maximum possible value infinity for the upper limit of y. Thus we will have the new chain "bottom", [1,1], [1,2], [1,inf] at which point it has already converged.

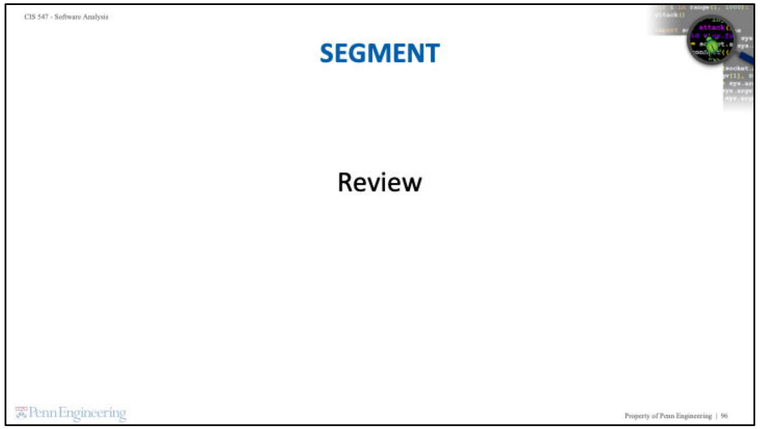
You can return to our example and convince yourself that using this solution would obtain the same result as the infinity-th iteration at the end of the 4th iteration.

The widening operator gets its name because it is an upper bound operator, and in many lattices, higher elements represent a wider range of program values. Similar to the other operations of a dataflow analysis, there are many choices in the design of a widening operator, which in turn affects the precision of the dataflow analysis.



LESSON

Review



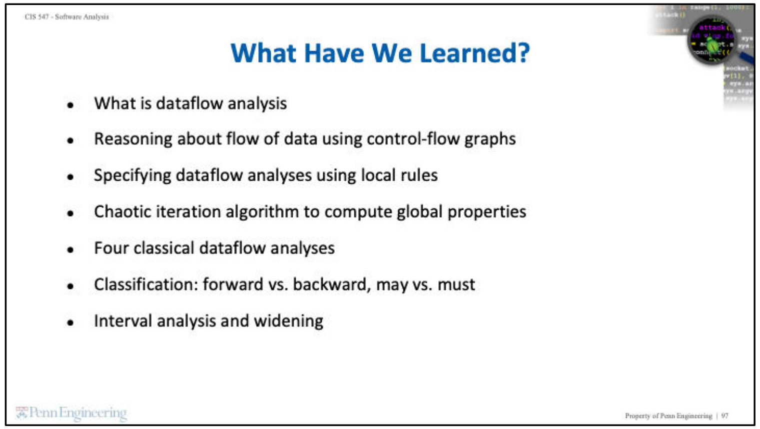
CIS 547 - Software Analysis

SEGMENT

Review

Penn Engineering

Property of Penn Engineering | 96



CIS 547 - Software Analysis

What Have We Learned?

- What is dataflow analysis
- Reasoning about flow of data using control-flow graphs
- Specifying dataflow analyses using local rules
- Chaotic iteration algorithm to compute global properties
- Four classical dataflow analyses
- Classification: forward vs. backward, may vs. must
- Interval analysis and widening

Penn Engineering

Property of Penn Engineering | 97

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

We introduced dataflow analysis, a common kind of static analysis that enables to reason about the flow of data in program runs.

We learnt how to reason about this flow of data using a program representation called a control-flow graph which concisely represents all runs of a program.

We saw how a dataflow analysis can be specified using local dataflow rules.

We learnt the chaotic iteration algorithm which repeatedly applies such rules to compute global dataflow properties.

We defined and saw examples of the four different classical dataflow analyses: Reaching Definitions Analysis, Very Busy Expressions Analysis, Available Expressions Analysis, and Live Variables Analysis.

We compared and contrasted these analyses along two important dimensions: forward vs. backward, and may vs. must.

And we finally learned about interval analysis, a modern dataflow analysis with many applications in security, and the concept of widening.

This module focused on how to reason about the flow of primitive data such as integers. In the next module, we will learn how to reason about the flow of non-primitive data, better known as pointers, objects, or references.