Program Testing and Analysis:

Program Slicing

Dr. Michael Pradel

Software Lab, TU Darmstadt

Warm-up Quiz

What does the following code print?

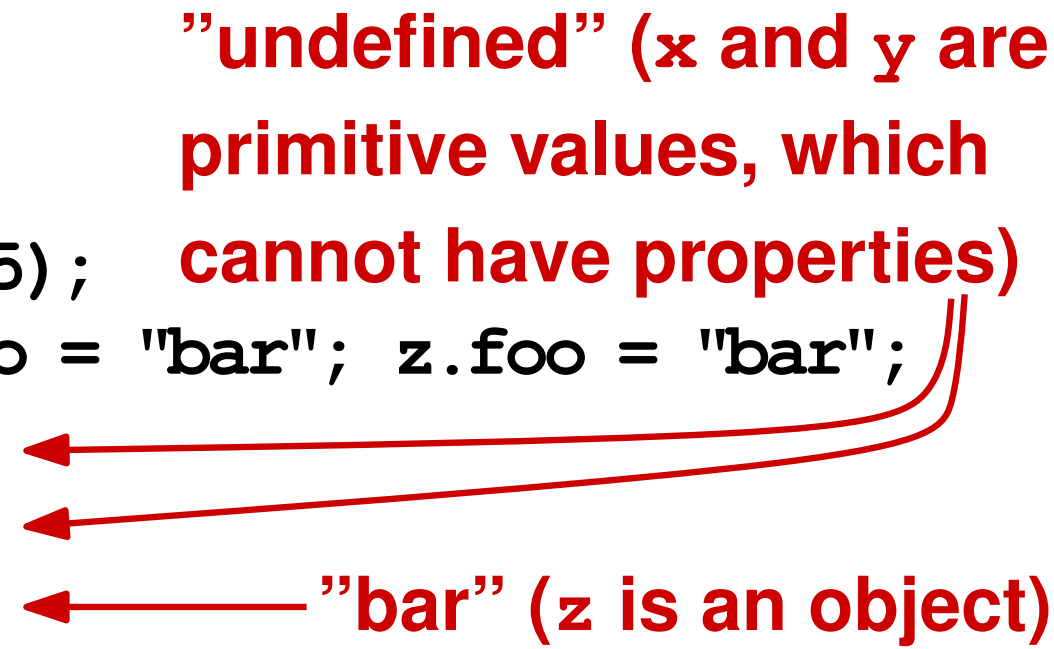
```
var x = 5;
var y = Number(5);
var z = new Number(5);
x.foo = "bar"; y.foo = "bar"; z.foo = "bar";
console.log(x.foo);
console.log(y.foo);
console.log(z.foo);
```

bar	undefined	bar	Some-
bar	undefined	undefined	thing
bar	undefined	bar	else

Warm-up Quiz

What does the following code print?

```
var x = 5;
var y = Number(5);
var z = new Number(5);
x.foo = "bar"; y.foo = "bar"; z.foo = "bar";
console.log(x.foo);
console.log(y.foo);
console.log(z.foo);
```



bar	undefined	bar	Some-
bar	undefined	undefined	thing
bar	undefined	bar	else

Outline

1. **Introduction**
2. **Static Slicing**
3. **Thin Slicing**
4. **Dynamic Slicing**

Mostly based on these papers:

- *Program Slicing*, Weiser., IEEE TSE, 1984
- *Thin Slicing*, Sridharan et al., PLDI 2007
- *Dynamic Program Slicing*, Agrawal and Horgan, PLDI 1990
- *A Survey of Program Slicing Techniques*, Tip, J Prog Lang 1995

Program Slicing

Extract an executable subset of a program that (potentially) affects the values at a particular program location

- Slicing criterion = program location + variable
- An observer focusing on the slicing criterion cannot distinguish a run of the program from a run of the slice

Example

```
var n = readInput();
var i = 1;
var sum = 0;
var prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i = i + 1;
}
console.log(sum);
console.log(prod);
```

Example

```
var n = readInput();
var i = 1;
var sum = 0;
var prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i = i + 1;
}
console.log(sum);
console.log(prod);
```

Slice for value of `sum` at this statement?

Example

```
var n = readInput();
var i = 1;
var sum = 0;
var prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i = i + 1;
}
console.log(sum);
console.log(prod);
```

Slice for value of `sum` at this statement?

Example

```
var n = readInput();
var i = 1;
var sum = 0;
var prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i = i + 1;
}
console.log(sum);
console.log(prod);
```

Slice for value of `prod` at this statement

Example

```
var n = readInput();
var i = 1;
var sum = 0;
var prod = 1;
while (i <= n) {
  sum = sum + i;
  prod = prod * i;
  i = i + 1;
}
console.log(sum);
console.log(prod);
```

Slice for value of `n` at this statement

Why Do We Need Slicing?

Various applications, e.g.

- Debugging: Focus on parts of program relevant for a bug
- Program understanding: Which statements influence this statement?
- Change impact analysis: Which parts of a program are affected by a change? What should be retested?
- Parallelization: Determine parts of program that can be computed independently of each other

Slicing: Overview

Forward vs. backward

- Backward slice (our focus): Statements that influence the slicing criterion
- Forward slice: Statements that are influenced by the slicing criterion

Static vs. dynamic

- Statically computing a minimum slice is undecidable
- Dynamically computed slice focuses on particular execution/input

Static Program Slicing

- **Introduced by Weiser**

  (IEEE TSE, 1984)

- **Various algorithms to compute slices**
- **Here: Graph reachability problem based on program dependence graph**

Program Dependence Graph

Directed graph representing the data and control dependences between statements

- Nodes:
  - Statements
  - Predicate expressions
- Edges:
  - Data flow dependences: One edge for each definition-use pair
  - Control flow dependences

Example: Data Flow Dependences

```
1  var n = readInput();
2  var i = 1;
3  var sum = 0;
4  var prod = 1;
5  while (i <= n) {
6    sum = sum + i;
7    prod = prod * i;
8    i = i + 1;
   }
9  console.log(sum);
10 console.log(prod);
```

Def \ Use	1	2	3	4	5	6	7	8	9	10
1					x					
2					x	x	x	x		
3						x			x	
4							x			x
5										
6						x			x	
7							x			x
8					x	x	x	x		
9										
10										

Control Flow Dependences

- Post-dominator:

  Node n_1 is post-dominated by node n_2 if every path $n_1, ..., exit$ in the control flow graph contains n_2

- Control dependence:

  Node n_2 is control-dependent on node n_1 if
  - there exists a control flow path $P = n_1, ..., n_2$ where any node in P (excluding n_1 and n_2) is post-dominated by n_2, and
  - n_1 is not post-dominated by n_2

Example: Post-dominators

```
1  var n = readInput();
2  var i = 1;
3  var sum = 0;
4  var prod = 1;
5  while (i <= n) {
6    sum = sum + i;
7    prod = prod * i;
8    i = i + 1;
   }
9  console.log(sum);
10 console.log(prod);
```

n_1 \ n_2	1	2	3	4	5	6	7	8	9	10
1		x	x	x	x				x	x
2			x	x	x				x	x
3				x	x				x	x
4					x				x	x
5									x	x
6					x		x	x	x	x
7					x			x	x	x
8					x				x	x
9										x
10										

Control dependences:

6 is control dependent on 5

7 — " — 5

8 — " — 5

Computing Slices

Given:

- Program dependence graph G_{PD}
- Slicing criterion (n, V), where n is a statement and V is the set of variables defined or used at n

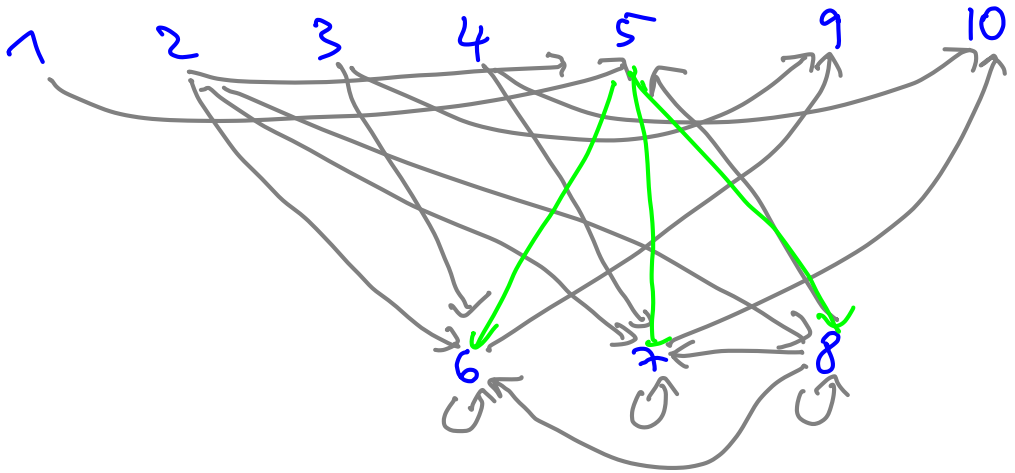
Slice for (n, V):

All statements from which n is reachable

(i.e., all statements on which n depends)

Example: Program Dep. Graph

```
1  var n = readInput();
2  var i = 1;
3  var sum = 0;
4  var prod = 1;
5  while (i <= n) {
6    sum = sum + i;
7    prod = prod * i;
8    i = i + 1;
   }
9  console.log(sum);
10 console.log(prod);
```



→ data dep.

→ control dep.

slice(9, {"sum"}) =

{n | reachable(n, 9)} =

{1, 2, 3, 5, 6, 8, 9}

Quiz

```
var x = 1;      // 1
var y = 2;      // 2
if (x < y) {    // 3
  y = x;        // 4
}
var z = x;      // 5
```

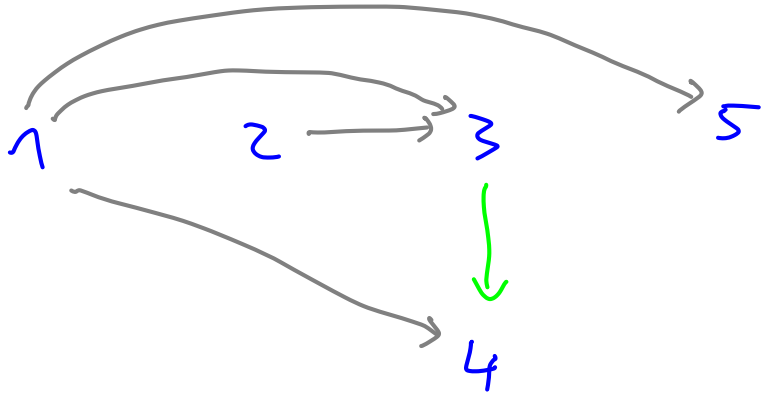
Draw the PDG and compute $slice(5, \{z\})$.

What is the sum of

- **the number of nodes,**
- **the number of edges, and**
- **the number of statements in slice?**

Quiz:

```
1   var x = 1
2   var y = 2
3   if (x < y)
4       y = x
5   var z = x
```



→ data

→ control

slice (5, {z}) = {1, 5}

⇒ Solution: 5 nodes + 5 edges + 2 stmts.

= 12

Outline

1. **Introduction**
2. **Static Slicing**
3. **Thin Slicing**
4. **Dynamic Slicing**

Mostly based on these papers:

- *Program Slicing*, Weiser., IEEE TSE, 1984
- *Thin Slicing*, Sridharan et al., PLDI 2007
- *Dynamic Program Slicing*, Agrawal and Horgan, PLDI 1990
- *A Survey of Program Slicing Techniques*, Tip, J Prog Lang 1995

Thin Slicing: Overview

- Challenge: Static slices are often very large
  - Worst case: Entire program
  - Too large for common debugging and program understanding tasks
- Main reason: Aims at an executable program
  - But: Not needed for many tasks
- Idea: Heuristically focus on statements needed for common debugging tasks
  $\rightarrow$ Thin slice
- Let user expand the thin slice on demand

Thin Slicing: Definition

- Statement directly uses a memory location if it uses it for some computation other than pointer dereference
  - Example: `x.f+y` uses `x` for pointer dereference and directly uses `y`
- Dependence graph G for thin slicing:
  Data dependences computed based on direct uses only
- Thin slice: Statements reachable from criterion's statement via G

Expanding Thin Slices

- Thin slices include ”producer statements” but exclude ”explainer statements”
  - Why do heap accesses read/write the same object?
  - Why can this producer execute?
- Most explainers are not useful for common tasks
- Expose explainers on demand via incremental expansion

Example: Thin slicing

```
1 var x = {};
2 var z = x;
3 var y = {};
4 var w = x;
5 w.f = y;
6 if (w === z) {
7   var v = z.f;   // criterion
  }
```

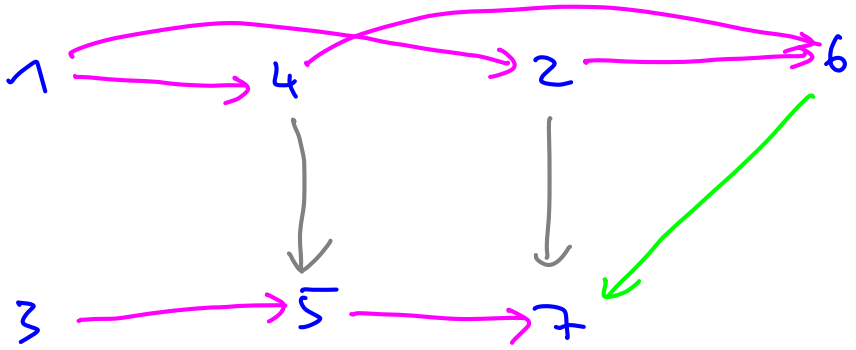
→ direct data dep.

→ data dep. for pointer dereferences

→ control dep.

} excluded

Dependence graph



- Traditional slice: All statements
- Thin slice: 3, 5, 7
- On demand expansion, e.g. "Why are w and z aliased?"

Evaluation and Results

- Simulate developer effort for bug finding
  - Set of known bugs that crash the program (and their root causes)
  - Assume that developer inspects statements with breadth-first search on PDG, starting from crash point
  - Count inspected statements with traditional and thin slice
- Results:
  - Mean of 12 inspected statements per thin slice
  - Overall, 3.3x fewer inspected statements

Outline

1. **Introduction**
2. **Static Slicing**
3. **Thin Slicing**
4. **Dynamic Slicing**

Mostly based on these papers:

- *Program Slicing*, Weiser., IEEE TSE, 1984
- *Thin Slicing*, Sridharan et al., PLDI 2007
- *Dynamic Program Slicing*, Agrawal and Horgan, PLDI 1990
- *A Survey of Program Slicing Techniques*, Tip, J Prog Lang 1995

Dynamic Slicing

- Various definitions

  Here: Agrawal & Horgan, PLDI 1990

- Dynamic slice: Statements of an execution that must be executed to give a variable a particular value
  - For an execution, i.e., a particular input
  - Slice for one input may be different from slice for another input

- Useful, e.g., for debugging: Get a reduced program that leads to the unexpected value

Dynamic Slice (Simple Approach)

- **Given: Execution history**
  - Sequence of PDG nodes that are executed
- **Slice for statement n and variable v:**
  - Keep PDG nodes only if there are in history
  - Use static slicing approach (= graph reachability) on reduced PDG

Example 1

```
var x = readInput();
if (x < 0) {
  y = x + 1;
  z = x + 2;
} else {
  if (x === 0) {
    y = x + 3;
    z = x + 4;
  } else {
    y = x + 5;
    z = x + 6;
  }
}
console.log(y);
console.log(z);
```

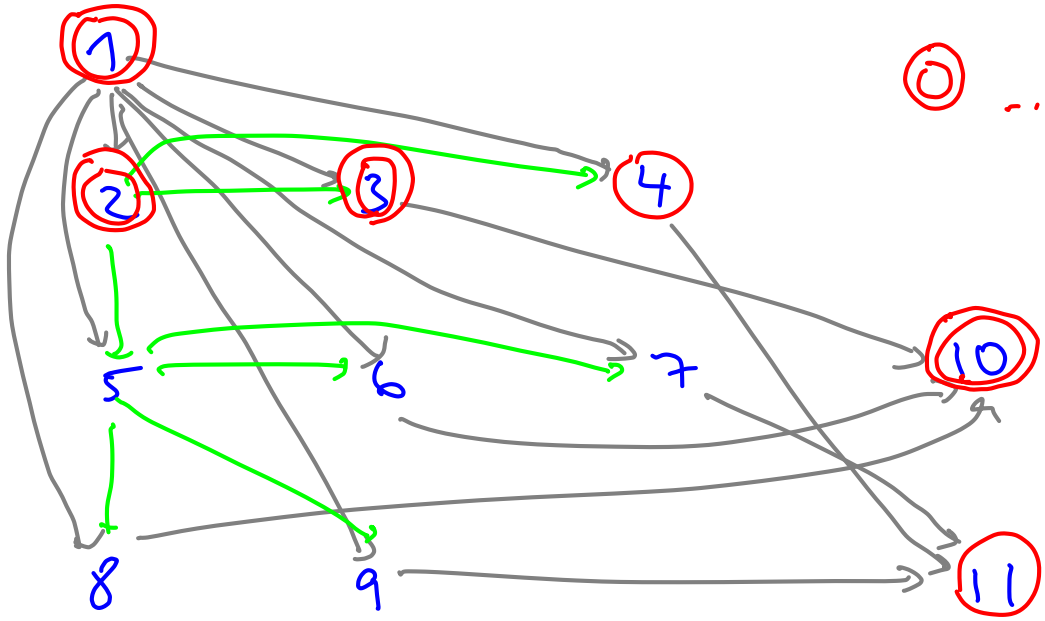
Example: Dynamic Slice (Simple Approach)

```
1   var x = readInput();
2   if (x < 0) {
3     y = x + 1;
4     z = x + 2;
    } else {
5     if (x === 0) {
6       y = x + 3;
7       z = x + 4;
      } else {
8       y = x + 5;
9       z = x + 6;
      }
    }
10  console.log(y);
11  console.log(z);
```

Input: −1

History: 1, 2, 3, 4, 10, 11

PDG:



1 .. executed

0 .. slice (10, {y})

Example 2: Quiz

```
var n = readInput(); // 1
var z = 0;           // 2
var y = 0;           // 3
var i = 1;           // 4
while (i <= n) {     // 5
  z = z + y;         // 6
  y = y + 1;         // 7
  i = i + 1;         // 8
}
console.log(z);      // 9
```

Example 2: Quiz

```
var n = readInput(); // 1
var z = 0;           // 2
var y = 0;           // 3
var i = 1;           // 4
while (i <= n) {     // 5
  z = z + y;         // 6
  y = y + 1;         // 7
  i = i + 1;         // 8
}
console.log(z);      // 9
```

Draw the PDG and compute the dynamic slice for statement 9 and variable `z`, with input `n=1`.

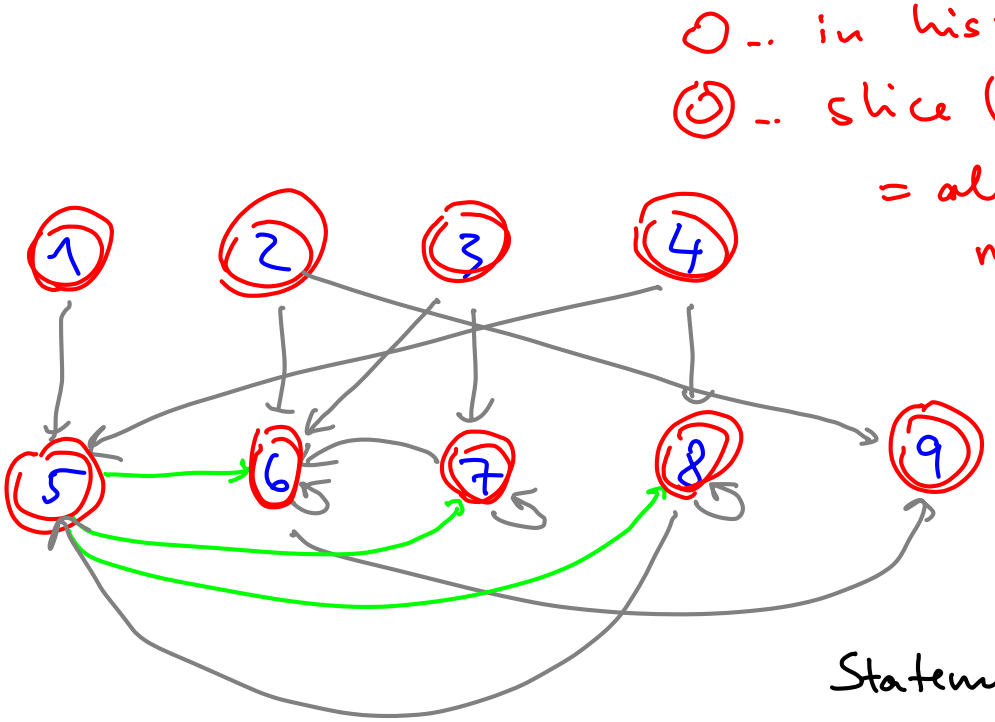
How many statements are in the slice?

Example 2

```
1  var n = readInput();
2  var z = 0;
3  var y = 0;
4  var i = 1;
5  while (i <= n) {
6    z = z + y;
7    y = y + 1;
8    i = i + 1;
   }
9  console.log(z);
```

Input: n = 1

History: 1, 2, 3, 4, 5, 6, 7, 8, 5, 9



○ .. in history

◎ .. slice (9, {z})

= all statements

BUT:

Statement 7 is not relevant!