Program Testing and Analysis:
Program Slicing (Part 2)

Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
function foo(a, a, a) {
    console.log(a);
}

foo("this", "that", "or maybe this");
```

"this"  "that"  "or maybe this"  Something else
Warm-up Quiz

What does the following code print?

```javascript
function foo(a, a, a) {
    console.log(a);
}

foo("this", "that", "or maybe this");
```

"this"  "that"  "or maybe this"  Something 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
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
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)
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
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
  → 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

```javascript
var x = {};  
var z = x;  
var y = {};  
var w = x;  
w.f = y;  
if (w === z) {
  var v = z.f; // criterion
}
```

→ direct data dep.
→ data dep. for pointer dereferences
→ control dep.

**Dependence graph**

1 → 4 → 2 → 6
3 → 5 → 7

*Traditional slice:*
All statement

*Thin slicing:*
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 they are in history
  - Use static slicing approach (= graph reachability) on reduced PDG
Example 1

```javascript
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)

```javascript
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);
```

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

PDG:

0 - executed
0 - slice (10, 11)

Criterion
Example 2: Quiz

```javascript
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

```javascript
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?
```javascript
var n = readInput();
var z = 0;
var y = 0;
var i = 1;
while (i <= n) {
    z = z + y;
    y = y + 1;
    i = i + 1;
}
console.log(z);
```

**Input:**  
\( n = 1 \)

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

- **0:** in history  
- **0:** slice (9, \{23\})  
  = entire program

But: statement 7 is irrelevant!
Limitations of Simple Approach

- **Multiple occurrences** of a single statement are represented as a single PDG node.

- **Difference occurrences** of a statement may have different dependences:
  - All occurrences get conflated.

- **Slices** may be larger than necessary.
Dynamic Slice (Revised Approach)

Dynamic dependence graph

- Nodes: Occurrences of nodes of static PDG
- Edges: Dynamic data and control flow dependences

Slice for statement $n$ and variables $V$ that are defined or used at $n$:

- Compute nodes $S_{dyn}$ that can reach any of the nodes that represent occurrences of $n$
- Slice = statements with at least one node in $S_{dyn}$
Example 2 (revised approach)

```
var n = readInput();
var z = 0;
var y = 0;
var i = 1;
while (i <= n) {
    z = z + y;
    y = y + 1;
    i = i + 1;
}
console.log(z);
```

Input:  \( n = 1 \)

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

O. slice (9, {+3})
Discussion: Dynamic Slicing

- May yield a program that, if executed, does not give the same value for the slicing criterion than the original program

- Instead: Focuses on isolating statements that affect a particular value
  - Useful, e.g., for debugging and program understanding

- Other approaches exist, see F. Tip’s survey (1995) for an overview
Summary

- **Program slicing**: Extract subset of statements for a particular purpose
  - Debugging, program understanding, change impact analysis, parallelization

- **Various techniques**
  - **Traditional static slicing**: Executable but potentially very large slice
  - **Thin slicing**: Focus on producer statements, reveal explainer statements on demand
  - **Dynamic slicing**: Useful for understanding behavior of particular execution