Program Testing and Analysis: Program Slicing

Prof. Dr. Michael Pradel
Software Lab, TU Darmstadt
Warm-up Quiz

What does the following code print?

```javascript
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?

```javascript
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);
```

"undefined" (x and y are primitive values, which cannot have properties)

"bar" (z is an object)
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

```javascript
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

```javascript
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

```javascript
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

```javascript
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

```javascript
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

```javascript
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);
```
Control Flow Dependences

- **Post-dominator:**
  Node $n_2$ (strictly) post-dominates node $n_1 (\neq n_2)$ if every path $n_1, ..., \text{exit}$ in the control flow graph contains $n_2$
Control Flow Dependences

- **Post-dominator:**
  Node $n_2$ (strictly) post-dominates node $n_1 (\neq n_2)$ if every path $n_1, ..., \text{exit}$ in the control flow graph contains $n_2$.

- **Control dependence:**
  Node $n_2$ is control-dependent on node $n_1 \neq n_2$ if
  - there exists a control flow path $P = n_1, ..., n_2$ where $n_2$ post-dominates any node in $P$ (excluding $n_1$), and
  - $n_2$ does not post-dominate $n_1$.
```javascript
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);
```

Control dependences:
6 is control-dep. 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)
```javascript
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);
```

\[\text{slice } (9, \{ \text{sum} \}) = \{ n \mid \text{reachable } (n, 9) \}\]
\[= \{ 1, 2, 3, 5, 6, 8, 9 \}\]
Quiz

```javascript
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 the slice?
1 var x = 1
2 var y = 2
3 if (x < y) {
4     y = x
5 }

\[ \text{Solution: 5 nodes + 5 edges + 2 stubs = 12} \]
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
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
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 de-reference  
- Control dep.

Dependence graph:

1 → 4 → 2 → 6  
3 → 5 → 7

- Traditional slice: all statements  
- 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 there 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);
```
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:

4
  ↓
2
  ↓
3
  ↓
4
  ↓
5
  ↓
6
  ↓
7
  ↓
10
  ↓
11

data
  └── control

O.. executed
O.. slice (10, [3])
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
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:** 1

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

- **Kernel:** 9
- **Active:** 9
- **In History:** 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Slice:** 9, {2}
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)

```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

O. slice (9, {?})
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