Program Analysis – Lecture 9
Program Slicing (Part 1)
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
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);
```
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
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
Variable Definition and Use

- A **variable definition** for a variable $v$ is a basic block that assigns to $v$
  - $v$ can be a local or global variable, parameter, or property

- A **variable use** for a variable $v$ is a basic block that reads the value of $v$
  - In conditions, computations, output, etc.
Definition-Clear Paths

A **definition-clear path** for a variable $v$ is a path $n_1, \ldots, n_k$ in the CFG such that

- $n_1$ is a variable definition for $v$
- $n_k$ is a variable use for $v$
- No $n_i$ (1 $< i \leq k$) is a variable definition for $v$
  - $n_k$ may be a variable definition if each assignment to $v$ occurs after a use

Note: Def-clear paths do not go from entry to exit (in contrast to our earlier definition of paths)
A definition-use pair (DU-pair) for a variable v is a pair of nodes \((d, u)\) such that there is a definition-clear path \(d, \ldots, u\) in the CFG.
```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 $n_1$
Example: Control Flow Dependences

```javascript
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);

Control dependences:
6 is control-dependent on 5
7
8
9
10
```
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 Dependence Graph

```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);
```

\[
\begin{align*}
\text{slice} (9, \{ \text{sum} \}) &= \{ n \mid \text{reachable} (n, 9) \} \\
&= \{ 1, 2, 3, 5, 6, 8, 9 \}
\end{align*}
\]