# Program Slicing

Zhiyuan Li

(Some examples and pages are converted from those produced by Mary Jean Harrold (GATech) and elsewhere

## Slicing – Introduction (2)

- A slice of a program wrt a program point p and set of variables v, consists of all statements and predicates in the program that may affect the values of the variables in v at p.
- Commonly computed using the Program Dependence Graph (PDG) and Systems Dependence Graph (SDG).
  - More about these later.

## Example – Program Slicing

- The essential idea of program slicing is to identify only those statements that are of relevance to the slicing criterion – i.e. those that affect or are affected by some statement.
- Slicing can move in two directions – forwards and backwards.
- A backwards slice identifies those statements that affect the computation of a particular statement.

Backward slice from `System.out.println(i);` :-

```
class Example{
  public static void main () {
      int sum = 0;
      int i = 1;
      while (i < 11) {
         sum = sum + i;
         i = i + 1;
      }
      System.out.println(sum);
      System.out.println(i);
   }
}
```

## Computation of Slice

- Backward slice can be easily computed from the PDG.
- Start at the node of interest and add it to the slice, and mark the node as visited.
- Trace back along the control and data dependencies, add these nodes to the slice, and mark them as visited.
- Any nodes marked as visited do not get visited again.
- Keep tracing back until the entry point of the program is reached.

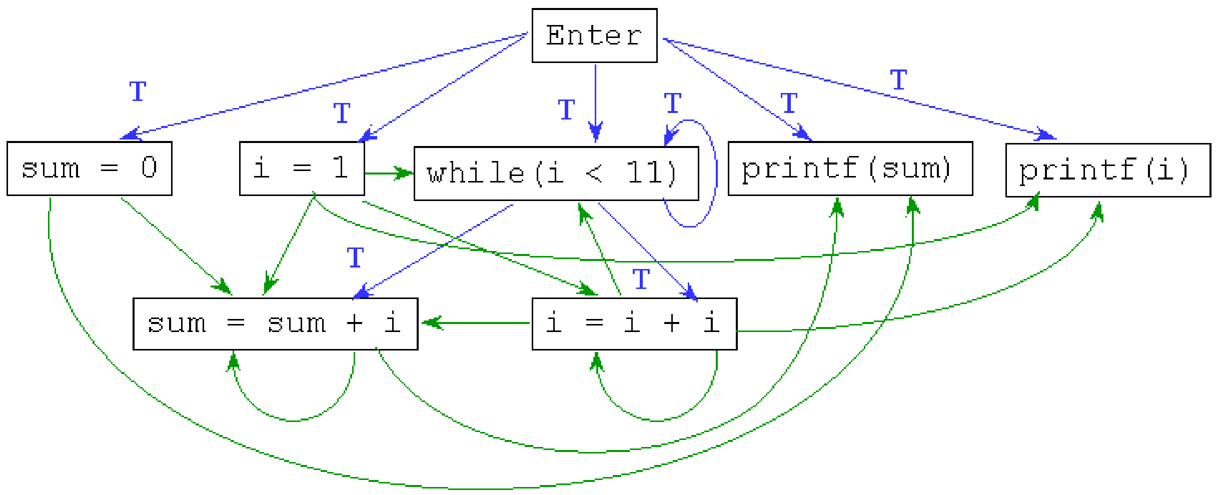
## Example – Computation of backward slice using PDG

Backward slice from `System.out.println(i);` :-

```
class Example{
  public static void main () {
      int sum = 0;
      int i = 1;
      while (i < 11) {
         sum = sum + i;
         i = i + 1;
      }
      System.out.println(sum);
      System.out.println(i);
   }
}
```

Control dependence

Flow dependence



## Forwards Slicing

- A forwards slice identifies those statements that are *affected by a particular statement*.
- Again computed from the PDG by following the control and data dependencies (forward).

Forward slice from the statement
`int sum = 0;` :-

```
class Example{
  public static void main () {
      int sum = 0;
      int i = 1;
      while (i < 11) {
         sum = sum + i;
         i = i + 1;
      }
      System.out.println(sum);
      System.out.println(i);
   }
}
```

## Inter-Procedural – Program Slicing

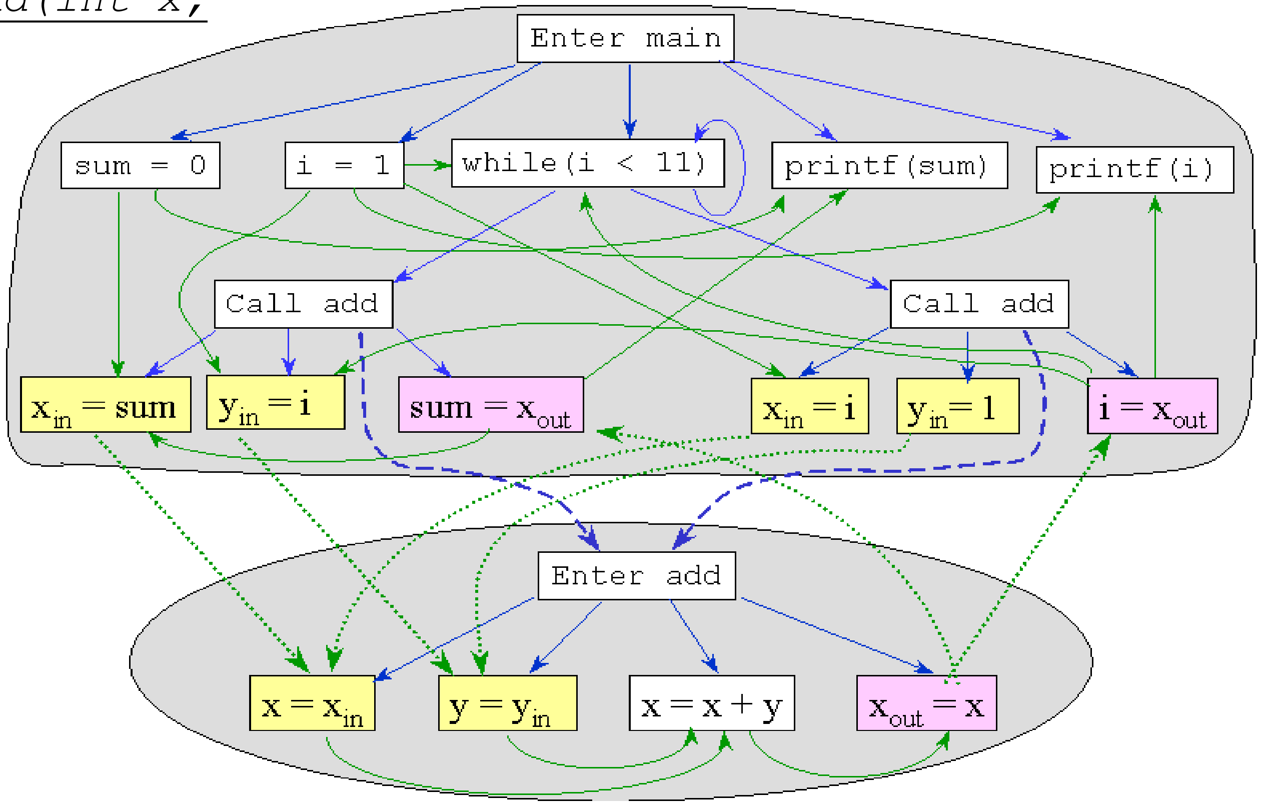
- Inter-procedural slices are those that incorporate method calls
- May also span class boundaries
- Computed in a similar fashion but using the SDG – need to follow control and data dependencies into other methods

Backward slice from `System.out.println(i);` :-

```
class Example{
  public static void main () {
     int sum = 0;
     int i = 1;
     while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
     }
     System.out.println(sum);
     System.out.println(i);
  }
  public static int add(int x, int y){
    return x + y;
  }
}
```

```
class Example{
  public static void main () {
      int sum = 0;
      int i = 1;
      while (i < 11) {
         sum = add(sum, i);
         i = add(i, 1);
      }
      System.out.println(sum);
      System.out.println(i);
  }
  public static int add(int x,
int y){
     return x + y;
  }
}
```

## Example – Computing Backward Slice from System.out.println(i);



## Extending this to OO - Attribute Slicing

- Slicing originally defined for and primarily applied to procedural programs.
- To apply it more generally to OO we need to modify the definition of a slice, since an attribute by itself does not constitute a target in the traditional meaning. Nor does a class have any notion of a start line or end line.
- If we consider the attribute itself to be the target then (informally):
  - A *backward slice* identifies those statements that affect the value of the attribute.
  - A *forward slice* identifies those statements that are affected by the value of the attribute.

# A Simple Example

<u>Original Class</u>

```
class A{
  int x;

  setx(int y){
    x = y;
  }

  getx(){
    return x;
  }

  incx(){
    x++
  }
}
```

<u>Backward Slice</u>

```
class A{
  int x;

  setx(int y){
    x = y;
  }

  incx(){
    x++
  }
}
```

<u>Forward Slice</u>

```
class A{
  int x;

  getx(){
    return x;
  }

  incx(){
    x++
  }
}
```

The method incx() appears in both cases since it both accesses and mutates the attribute.

Note that the backward slice essentially extracts the mutators and the forward slice the accessors.

By performing both backwards and forwards slicing the entire class is returned

## Slicing at Different Abstraction Levels

- In the example shown the slices have returned the entire methods (because they are simple)
- Attribute slicing can be applied at difference levels of abstraction:
  - Method level: the *entire* method is included if the method participates in the slice in any way (this can be computed on dataflow information alone)
  - Intra-method level: only the method code relevant to the attribute is returned thereby yielding more precise information (this requires structural information and dataflow information)

## Intra-Method Slicing Example

An intra-method backward slice on balance returns only the highlighted code relevant to the balance attribute (forwards and backwards slices are identical in this case, but not generally)

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    // constructor and other methods omitted
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount: " +
                                  amount);
        }
    }
}
```

## Backward Slicing (Ex. 2)

Use Criterion<10, product>

```
1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do {
6. sum := sum + i
7. product := product * i
8. i := i + 1 }
9. write (sum)
10. write (product)
```

```
1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do {
6.      sum := sum + i
7.      product := product * I
8.      i := i + 1  }
9. write (sum)
10. write (product)
```

# New Executable

```
1. read (n)
2. i := 1
3.
4. product := 1
5. while i <= n do {
6.
7.      product := product * I
8.      i := i + 1  }
9.
10. write (product)
```

## Forward Slicing (Ex. 2)
## Use Criterion <3, sum>

```
1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do {
6. sum := sum + i
7. product := product * i
8. i := i + 1 }
9. write (sum)
10. write (product)
```

```
1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do {
6. sum := sum + i
7. product := product * i
8. i := i + 1 }
9. write (sum)
10. write (product)
```

# Forward Slice (Ex. 3)
# Use Criterion <1, n>

```
1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do {
6.    sum := sum + i
7.    product := product * i
8.    i := i + 1 }
9. write (sum)
10. write (product)
```

- These forward slices are not executable.
- To make executable forward slices, more statements need to be added.

## Dynamic Slicing

- Include all those statements executed given a set of input
- It requires run-time recording of the execution trace
- Trace analysis is potentially very expensive

# Dynamic Backward Slicing (Ex. 1)

for initial values n=1, c1=true, c2= true

Using Criterion <12, z>

```
1. read (n)
2. for I := 1 to n do {
3.   a := 2
4.   if c1 then {
5.      if c2 then
6.         a := 4
7.      else
8.         a := 6 }
9.   z := a
10. c1 := true
11. c2 := true }
12. write (z)
```

Execution history is
1[1], 2[1], 3[1], 4[1], 5[1],
6[1], 9[1], 10[1], 11[1]
2[2], 12[1]

Note: 1[1] means first instance of statement 1

```
1. read (n)
2. for I := 1 to n do {
3.   a := 2
4.   if c1 then {
5.      if c2 then
6.         a := 4
7.      else
8.         a := 6 }
9.   z := a
10. c1 := true
11. c2 := true }
12. write (z)
```

Execution history is
1[1], 2[1], 3[1], 4[1], 5[1], 6[1], 9[1], 10[1], 11[1] 2[2], 12[1]

Note: 1[1] means first instance of statement 1

# Dynamic Backward Slicing (Ex. 2)

for initial values n=2, c1=false, c2= false

Using Criterion <12, z>

```
1. read (n)
2. for I := 1 to n do {
3.   a := 2
4.   if c1 then {
5.      if c2 then
6.         a := 4
7.      else
8.         a := 6 }
9.    z := a
10.  c1 := true
11.  c2 := true  }
12. write (z)
```

Execution history is
1[1], 2[1], 3[1], 4[1], 9[1], 10[1], 11[1], 2[2], 3[2], 4[2], 5[1], 6[1], 9[2],  10[2], 11[2], 2[3], 12[1]>

```
1. read (n)
2. for I := 1 to n do {
3.   a := 2
4.   if c1 then {
5.      if c2 then
6.         a := 4
7.      else
8.         a := 6 }
9.    z := a
10.  c1 := true
11.  c2 := true  }
12. write (z)
```

Execution history is
1[1], 2[1], 3[1], 4[1], 9[1], 10[1], 11[1], 2[2], 3[2], 4[2], 5[1], 6[1], 9[2], 10[2], 11[2], 2[3], 12[1]>

How do we compute the dynamic backward slice?