Program Slicing

Xiangyu Zhang

Outline

- What is slicing.
- Why slicing.
- Static slicing.
- Dynamic slicing.
  - Data dependence detection
  - Control dependence detection
  - Slicing algorithms (forward vs. backward)
  - Chopping

What is a slice?


```
1  Void main ( ) {
2    int I=0;
3    int sum=0;
4    while (I<N) {
5        sum=add(sum,I);
6        I=add(I,1);
7    }
8    printf ("sum=%d\n",sum);
9    printf("I=%d\n",I);
```

S: …. = f (v)

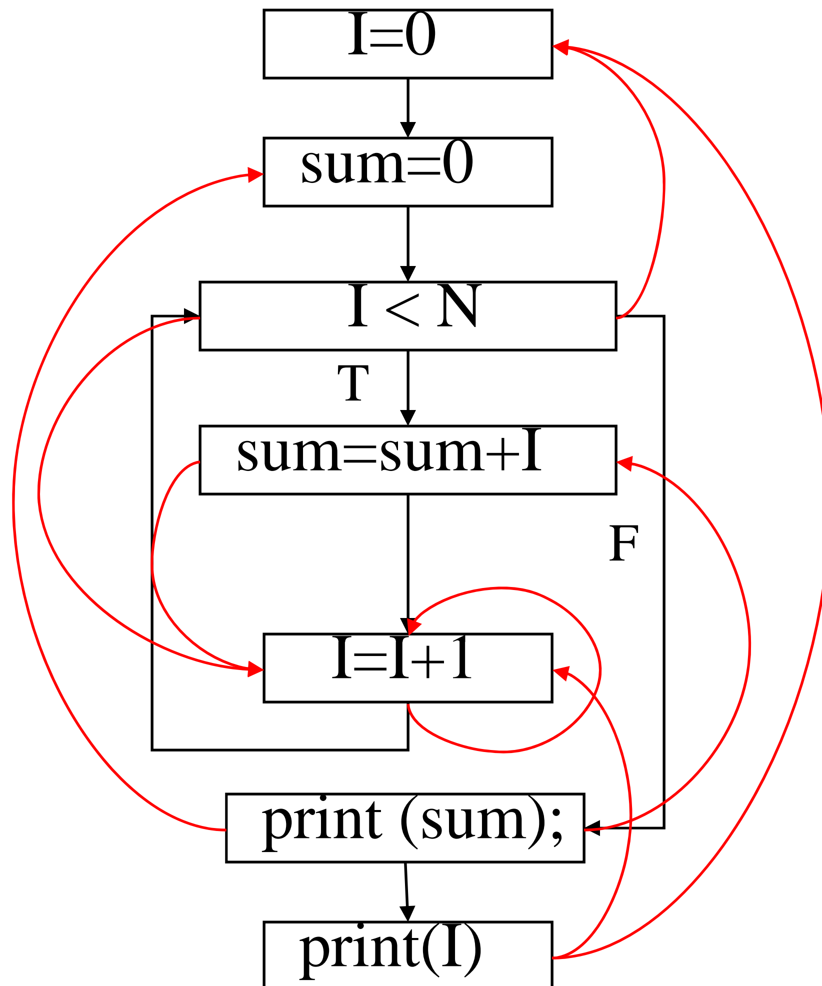
- Slice of v at S is the set of statements involved in computing v's value at S.

[Mark Weiser, 1982]

Why Slicing?

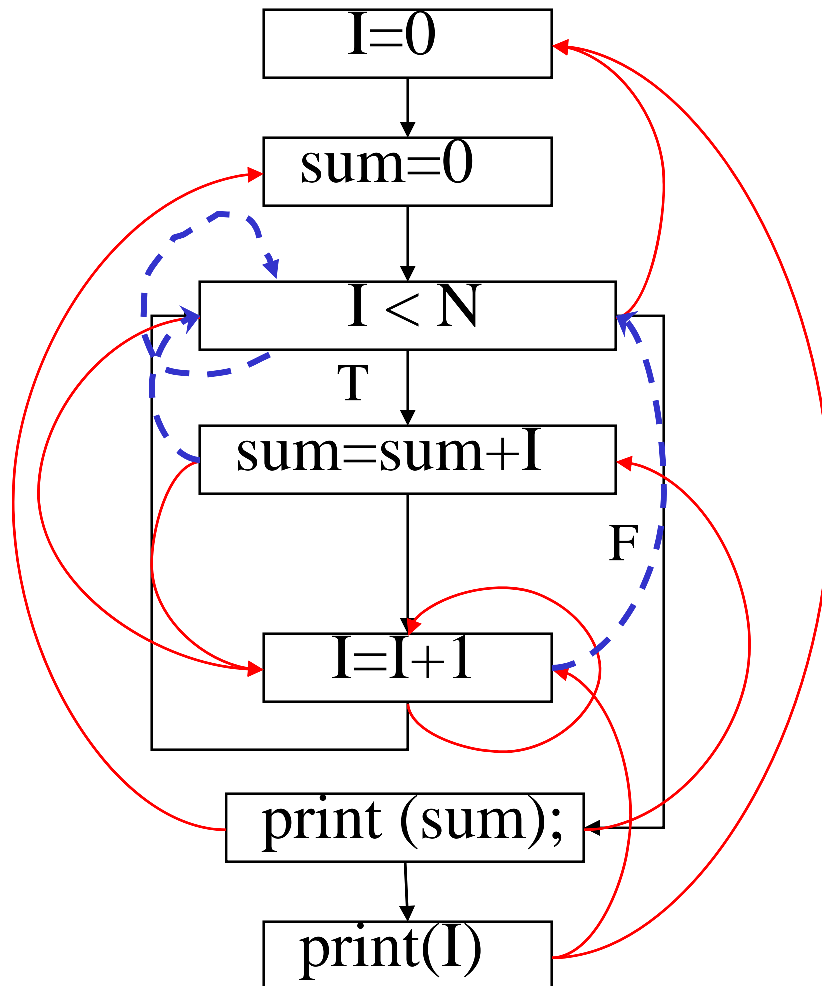
- *Debugging:* that's why slicing was introduced.
- *Data Flow Testing:* Reduce cost of regression testing after modifications to the program.
- *Code Reuse:* Extracting modules for reuse.
- *Version Integration:* Safely integrate non-interfering extensions of an application.
- *Partial Execution replay:* Replay only part of the execution that is relevant to a failure.
- *Partial roll back:* partially roll back a transaction.
- *Information flow:* prevent confidential information from being sent out to untrusted environment.
- Others.

How to Compute Slices?



- Dependence Graph
  - Data dep.
  - Control dep.
- X is data dependent on Y if (1) there is a variable v that is defined at Y and used at X and (2) there exists a path of nonzero length from Y to X along which v is not re-defined.

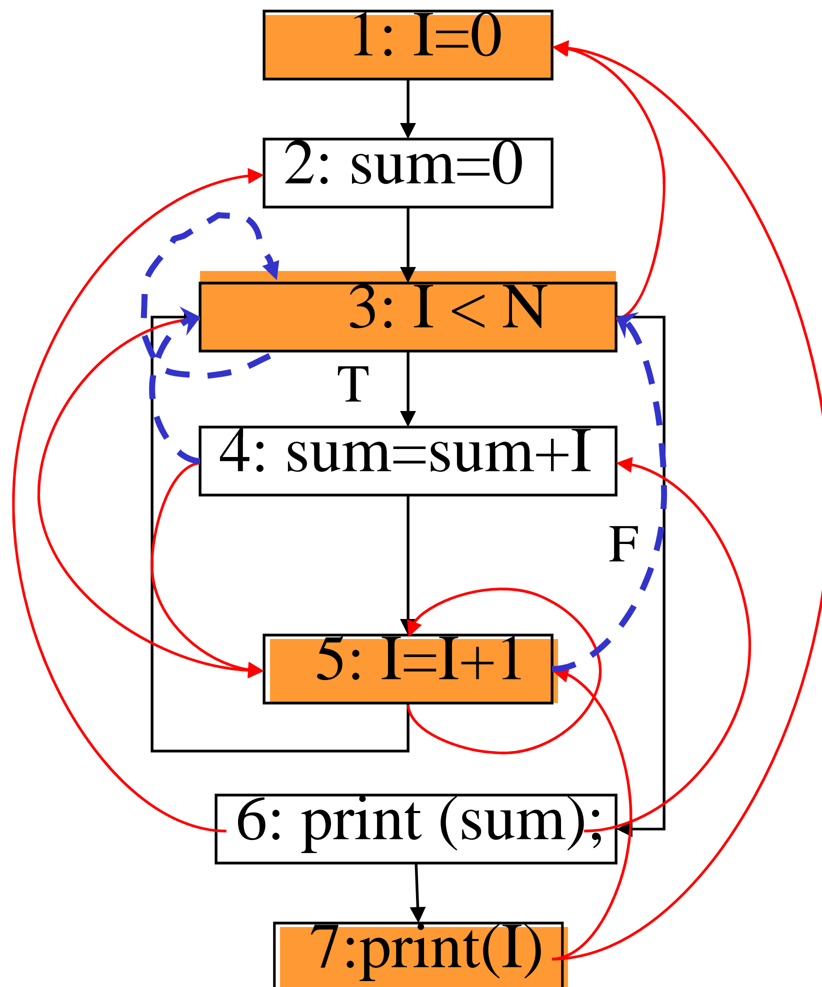
How to Compute Slices? (continued)



- Dependence Graph
  - Data dep.
  - Control dep.

- Y is control-dependent on X iff X directly determines whether Y executes
  - X is not strictly post-dominated by Y
  - there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

How to Compute Slices? (continued)



- Given a slicing criterion, i.e., the starting point, a slice is computed as the set of reachable nodes in the dependence graph

Slice(I@7)={1,3,5,7}

Slice(6)=?

Static Slices are Imprecise

- Don't have dynamic control flow information

```
1: if (P)
2:    x=f(…);
3: else
4:    x=g(…);
5. …=x;
```

- Use of Pointers – static alias analysis is very imprecise

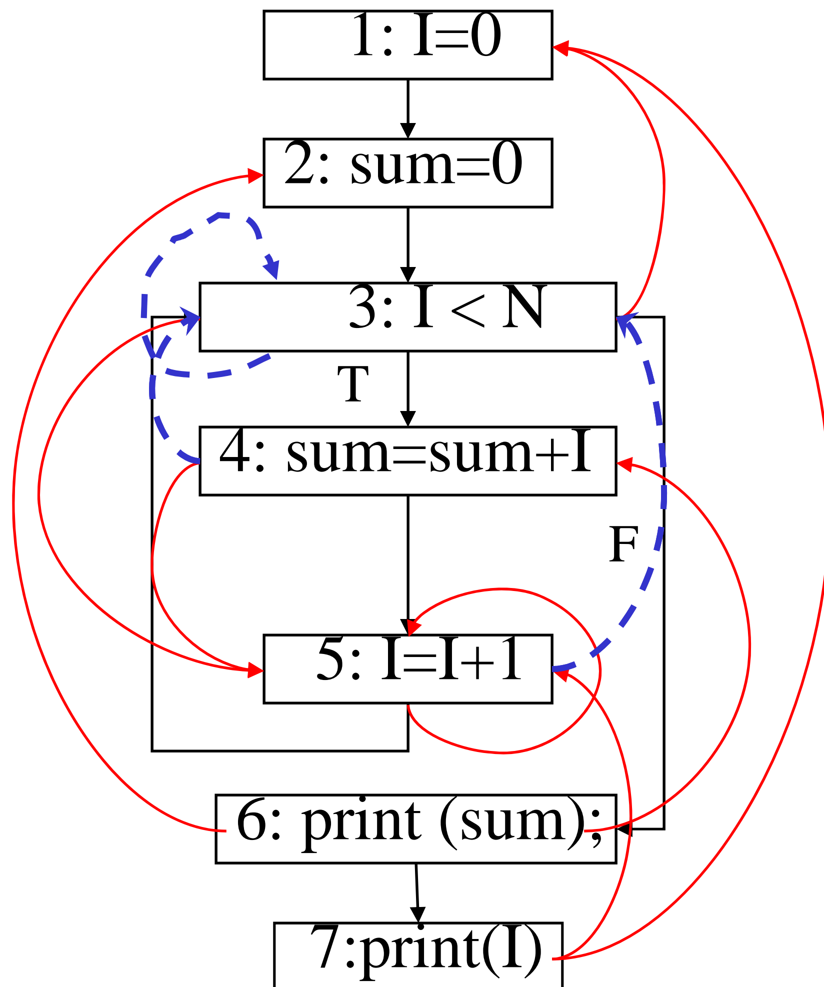
```
1: int a,b,c;
2: a=…;
3: b=…;
4: p=&a;
5: …=p[i];
```

- Use of function pointers

Dynamic Slicing

- Korel and Laski, 1988
  - The set of executed statement instances that did contribute to the value of the criterion.
- Dynamic slicing makes use of all information about a particular execution of a program.
- Dynamic slices are often computed by constructing a dynamic program dependence graph (DPDG).
  - Each node is an executed statement (instruction).
  - An edge is present between two nodes if there exists a data/control dependence.
  - A dynamic slice criterion is a triple <Var, Execution Point, Input>
  - The set of statements reachable in the DPDG from a criterion constitute the slice.
- Dynamic slices are smaller, more precise, more helpful to the user

An Example



Slice(I@7)={1,3,5,7}

Trace (N=0)

1_1: I=0

2_1: sum=0

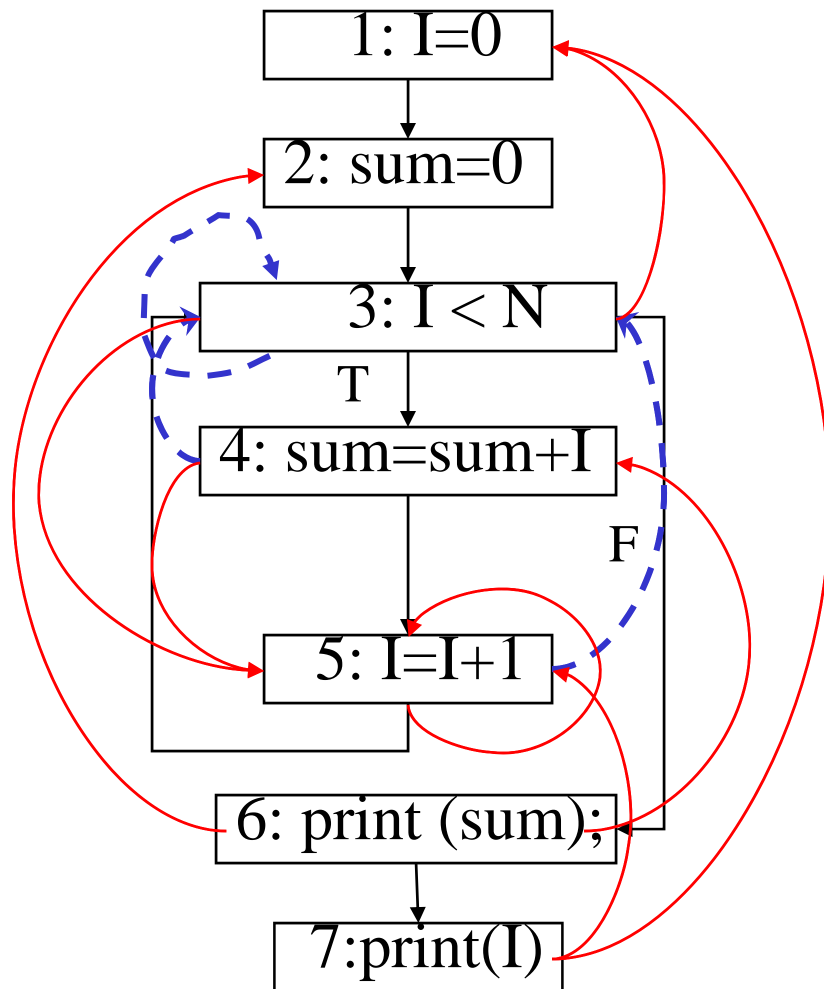
3_1: I<N

6_1: print(sum)

7_1: print(I);

DSlice(I@7_1,,N=0)={1,7}

Another Example



Slice(I@7)={1,3,5,7}

Trace (N=1)

1_1: I=0

2_1: sum=0

3_1: I<N

4_1: sum=sum+I

5_1: I=I+1

3_2: I<N

6_1: print(sum)

7_1: print(I);

DSlice(I@7_1,N=1)={1,3,5,7}

Effectiveness of Dynamic Slicing

Program	Static / Dynamic (25 slices)		
	AVG	MIN	MAX
126.gcc	5448	3.5	27820
099.go	1258	2	4246
134.perl	66	1	1598
130.li	149	1	1436
008.espresso	49	1	1359

- Sometimes, static and dynamic get the same answers.
- Sometimes, static gets explosions
- On average, static slices can be many times larger

Computing Dynamic Slices

- Data dependence: - do we still care about aliasing?
  - No.
  - A backwards linear scan over the trace is able to recover all data dependences.
- Control dependence
  - In order to find the predicate instance that a statement execution X is control dependent on, can I simply traverse backwards and find the closest predicate?
    - No.
  - There exist inefficient backwards algorithms

Trace:
1_1: if (P)
2_1: …
3_1: i=…

Offline Algorithms – Data Dep

- Instrument the program to generate the control flow and memory access trace

```
    Void main ( ) {
1     int I=0;
2     int sum=0;
3     while (I<N) {
4         sum=add(sum,I);
5         I=add(I,1);
6     }
7     printf ("sum=%d\n",sum);
8     printf("I=%d\n",I);
```

Offline Algorithms – Data Dep

- Instrument the program to generate the control flow and memory access trace

```
    Void main ( ) {
1       int I=0; trace("1 W  "+&I);
2       int sum=0; trace("2 W "+&sum);
3       while (trace("3 R "+&I+&N),I<N) {
4           sum=add(sum,I);
            trace("4 R "+&I+&sum+ " W "
                    +&sum);
5           I=add(I,1);
6       }
7       printf ("sum=%d\n",sum);
8       printf("I=%d\n",I);
```

Trace (N=0)

```
1 W  &I
2 W  &sum
3 R &I  &N
4 R &I &sum  W &sum
5 R &I   W &I
3 R &I  &N
7 R &sum
8 R &I
```

Offline Algorithms – Data Dep

- Instrument the program to generate the control flow and memory access trace
- For a "R, addr", traverse backward to find the closest "W,addr", introduce a DD edge, traverse further to find the corresponding writes of the reads on the identified write.
  - "8, R &I" -> "5, W &I"-> "5, R &I"->"1, R&I"

Trace (N=0)

1 W &I
2 W &sum
3 R &I &N
4 R &I &sum W &sum
5 R &I W &I
3 R &I &N
7 R &sum
8 R &I

Offline Algorithms – Control Dep

- Assume there are no recursive functions and CD(i) is the set of static control dependence of i, traverse backward, find the closest x, s.t. x is in CD(i), introduce a dynamic CD from i to x.

- Problematic in the presence of recursion.

Efficiently Computing Dynamic Dependences

- The previous mentioned graph construction algorithm implies offline traversals of long memory reference and control flow traces
- Efficient online algorithms
  - Online data dependence detection.
  - Online control dependence detection.

Efficient Data Dependence Detection

- Basic idea

  i: x=… => hashmap[x]= i

  j: … =x… => dependence detected j → hashmap[x], which is j→i

Trace (N=1)	HashMap	Data Dep.
1_1: I=0	I: 1_1	
2_1: sum=0	I: 1_1 sum: 2_1	
3_1: $I<N$		3_1 →hashmap[I]=1_1
4_1: sum=sum+I	I: 1_1 sum: 4_1	4_1 →hashmap[sum]=2_1
5_1: I=I+1	I: 5_1 sum: 4_1	5_1 →hashmap[I]=1_1
3_2: $I<N$		3_2 →hashmap[I]=5_1
6_1: print(sum)		6_1 →hashmap[sum]−4_1
7_1: print(I);		7_1 →hashmap[I]=5_1

Efficient Dynamic Control Dependence (DCD) Detection

- **Def**: y_j DCD on x_i iff there exists a path from x_i to Exit that does not pass y_j and no such paths for nodes in the executed path from x_i to y_j.
- **Region**: executed statements between a predicate instance and its immediate post-dominator form a region.

Region Examples

```
1. for(i=0; i<N, i++) {
2.      if(i%2 == 0)
3.           p = &a[i];
4.      foo(p);
5. }
6. a = a+1;
```

- A statement instance x_i DCD on the predicate instance leading x_i 's enclosing region.
- Regions are either nested or disjoint. Never overlap.

1_1. for(i=0; i<N, i++) {
2_1. if(i%2 == 0)
3_1. p = &a[i];
4_1. foo(p);
…
1_2. for(i=0; i<N, i++) {
2_2. if(i%2 == 0)
4_2. foo(p);
…
1_3. for(i=0; i<N, i++) {
6_1. a = a+1;

DCD Properties

- **Def**: y_j DCD on x_i iff there exists a path from x_i to Exit that does not pass y_j and no such paths for nodes in the executed path from x_i to y_j.
- **Region**: executed statements between a predicate instance and its immediate post-dominator form a region.
- Property One: A statement instance x_i DCD on the predicate instance leading x_i 's enclosing region.

Property One

- A statement instance x_i DCD on the predicate instance leading x_i's enclosing region.

  Proof: Let the predicate instance be p_j and assume x_i does not DCD p_j. Therefore,

  either there is not a path from p_j to exit that does not pass x_i, which indicates x_i is a post-dominator of p_j, contradicting the condition that x_i is in the region delimited by p_j and its immediate post-dominator;

  or there is a y_k in between p_j and x_i so that y_k has a path to exit that does not pass x_i. Since p_j's immediate post-dominator is also a post dominator of y_k, y_k and p_j's post-dominator form a smaller region that include x_i, contradicting that p_j leads the enclosing region of x_i.

DCD Properties

- **Def**: y_j DCD on x_i iff there exists a path from x_i to Exit that does not pass y_j and no such paths for nodes in the executed path from x_i to y_j.
- **Region**: executed statements between a predicate instance and its immediate post-dominator form a region.
- Property Two: regions are disjoint or nested, never overlap.

Property Two

- Regions are either nested or disjoint, never overlap.

  Proof: Assume there are two regions (x, y) and (m, n) that overlap. Let m reside in (x, y). Thus, y resides in (m, n), which implies there is a path from m to exit without passing y. Let the path be P. Therefore, the path from x to m and P constitute a path from x to exit without passing y, contradicting the condition that y is a post-dominator of x.

Efficient DCD Detection

- **Observation**: regions have the LIFO characteristic.
  - Otherwise, some regions must overlap.
- **Implication:** the sequence of nested active regions for the current execution point can be maintained by a stack, called control dependence stack (CDS).
  - A region is nested in the region right below it in the stack.
  - The enclosing region for the current execution point is always the top entry in the stack, therefore the execution point is control dependent on the predicate that leads the top region.
  - An entry is pushed onto CDS if a branching point (predicates, switch statements, etc.) executes.
  - The current entry is popped if the immediate post-dominator of the branching point executes, denoting the end of the current region.

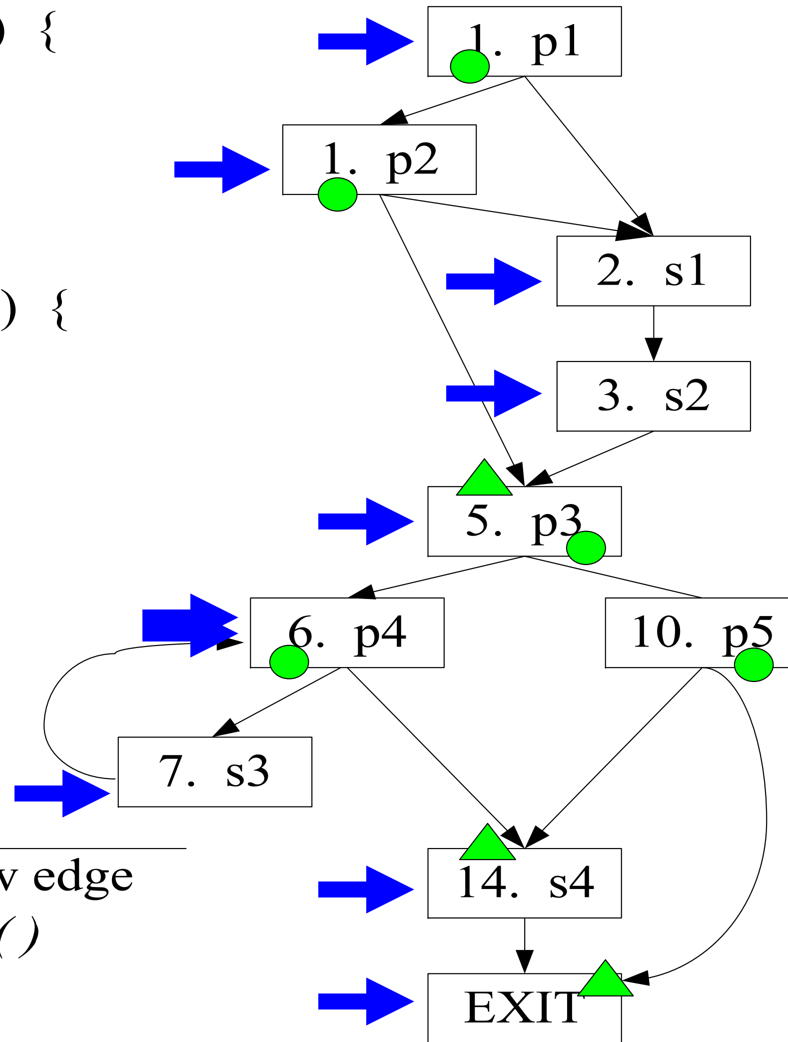
Algorithm

```
Predicate (x_i)
{
    CDS.push(<x_i, IPD(x) >);
}

Merge (t_j)
{
     while (CDS.top( ).second==t)
         CDS.pop( );
}
GetCurrentCD ( )
{
    return CDS.top( ).first;
}
```

An Example


```
1.   if ( p1 || p2 ) {
2.       s1;
3.       s2;
4.   }
5.   if (p3) {
6.       while (p4) {
7.           s3;
8.       }
9.   } else {
10.      if (p5) {
11.          return;
12.      }
13.  }
14.  s4;
```



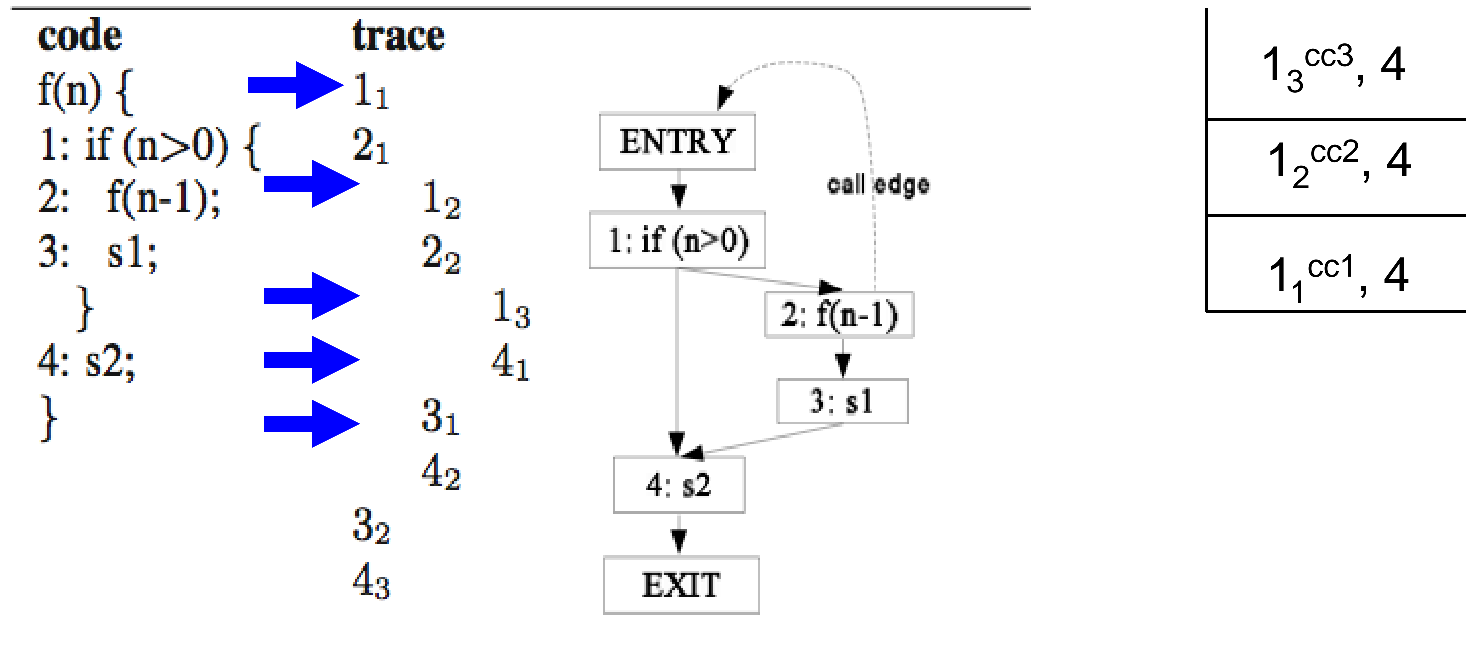
→	control flow edge
●	*Branching ( )*
▲	*Merging ( )*

6_2, 14

p2@14, 5

p5@EXIT

Interprocedural Control Dependence



- Annotate CDS entries with calling context.

Wrap Up

- We have introduced the concept of slicing and dynamic slicing
- Offline dynamic slicing algorithms based on backwards traversal over traces is not efficient
- Online algorithms that detect data and control dependences are discussed.

Forward Dynamic Slice Computation

- The approaches we have discussed so far are backwards.
  - Dependence graphs are traversed backwards from a slicing criterion.
  - The space complexity is O (execution length).
- Forward computation
  - A slice is represented as a set of statements that are involved in computing the value of the slicing criterion.
  - A slice is always maintained for a variable.

The Algorithm

- An assignment statement execution is formulated as
  - s_i: $x = p_j ? op(src1, src2, \ldots)$;
    - That is to say, the statement execution instance s_i is control dependent on p_j and operates on variables of src1, src2, etc.
- Upon the execution of s_i, the slice of x is updated to
  - $Slice(x) = \{s\} \cup Slice(src1) \cup Slice(src2) \cup \ldots \cup Slice(p_j)$
    - The slice of variable x is the union of the current statement, the slices of all variables that are used and the slice of the predicate instance that s_i is control dependent on. Because they are all contributing to the value of x.
    - Such slices are equivalent to slices computed by backwards algorithms.
      - Proof is omitted.
    - Slices are stored in a hashmap with variables being the keys.
    - The computation of Slice (p_j) is in the next slide. Note that p_j is not a variable.

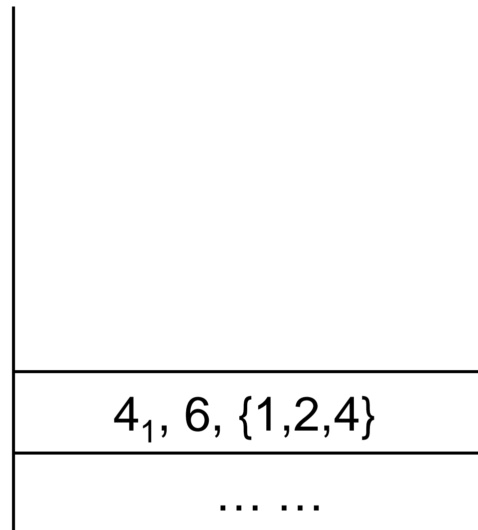
The Algorithm (continued)

- A predicate is formulated as
  - s_i: p_j? op (src1, src2, …)
    - That is to say, the predicate itself is control dependent on another predicate instance p_j and the branch outcome is computed from variables of src1, src2, etc.
- Upon the execution of s_i
  - A triple is pushed to CDS with the format of
    - <s_i, IPD(s), s U Slice (src1) U Slice (src2) U… U Slice(p_j) >
  - The entry is popped at its immediate post dominator
- Slice(p_j) can be retrieved from the top element of CDS.

Example

```
1:      a=1
2:      b=2
3:      c=a+b
4:      if a<b then
5:          d=b*c
6:          ........
```

Statements Executed	Dynamic Slices
1_1: a=1	Slice(a) = {1}
2_1: b=2	Slice(b) = {2}
3_1: c=a+b	Slice(c) = {1,2,3}
4_1: if a<b then	push(<4_1,6, {1,2,4}>)
5_1: d=b*c	Slice(d) = {1,2,3,4,5}



Properties

- The slices are equivalent to those computed by backwards algorithms
  - The proof is omitted.
- The space complexity is bounded
  - O ((# of variables + MAX_CDS_DEPTH) * # of statements)
- Efficiency relies on the hash map implementation and set operations.
  - A cost-effective implementation will be discussed later in cs510.

Extending Slicing

- Essentially, slicing is an orthogonal approach to isolate part of a program (execution) giving certain criterion.
- Mutations of slicing
  - Event slicing – intrusion detection, execution fast forwarding, understanding network protocol, malware replayer.
  - Forward slicing.
  - Chopping.
  - Probabilistic slicing.