Program Slicing (Part 2)

Shrawan Kumar

Senior Scientist @ TCS Research

shrawan.kumar@tcs.com

Agenda

- Variants of static slicing
- Challenges in static slicing
- Dynamic slicing
- Slicing variants in recent past
- References

To recapitulate …

- Executable backward static slicing by Weiser (as program slicing)
  - Slice P’ = S(P, <s, X>) is result of deleting some statements from P, such that

    P and P’ are indistinguishable when their behaviour is observed through “same corresponding window”, <s,X> and <s’,X> respectively

    For all corresponding terminating executions of P and P’ (on same input)
    Value of X at s in execution of P is same as value of X at s’ in execution of P’

- Computed as a data flow equation or by backward traversal over PDG
  - PDG is a graph representing data dependence as well as control dependence relationship among statements

- A minimal slice is non-computable

Variants of static program slicing

- Backward / Forward
- Executable / Non-executable
- Intraprocedural / Interprocedural

Forward slicing

- To answer the question

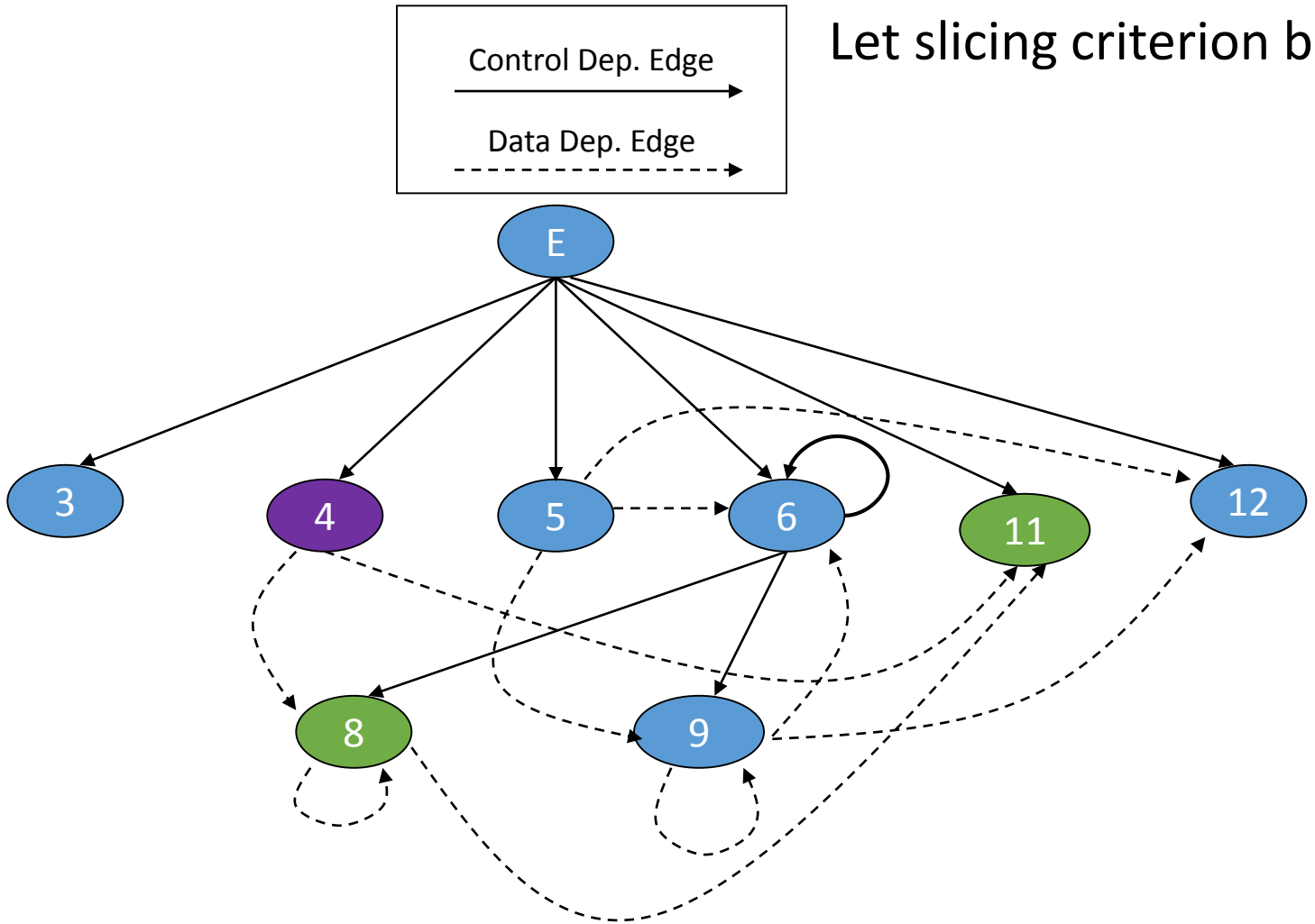
  Which statements get influenced if value of v is changed at *s*

- The general slicing criterion is still <*s*, X>

- Forward slice with respect to <*s*, X> consists of all statements and predicates that may be affected by the value of X at *s*

- Forward slice is non-executable, in general

Forward slicing using PDG



Let slicing criterion be <4,{sum}>

```
1.  main()
2.  {
3.    int i, sum;
4.    sum = 0
5.    i= 1
6.    while (i<=10)
7.    {
8.      sum = sum + 1
9.      i = i + 1
10.   }
11.   printf("%d", sum)
12.   printf("%d\n", i)
13. }
```

Nodes which are reachable (forward traversal) from slicing criterion node are in slice : {8,11}

Use of forward slicing

- Maintenance
  - Identifying impacted piece of code for a change in code
- Regression testing
  - Identifying minimal set of test cases to run for incremental changes

Challenges in static slicing

- Unstructured programs
- Arrays
- Pointers
- Structures
- Procedures
- Parallel programs
- …

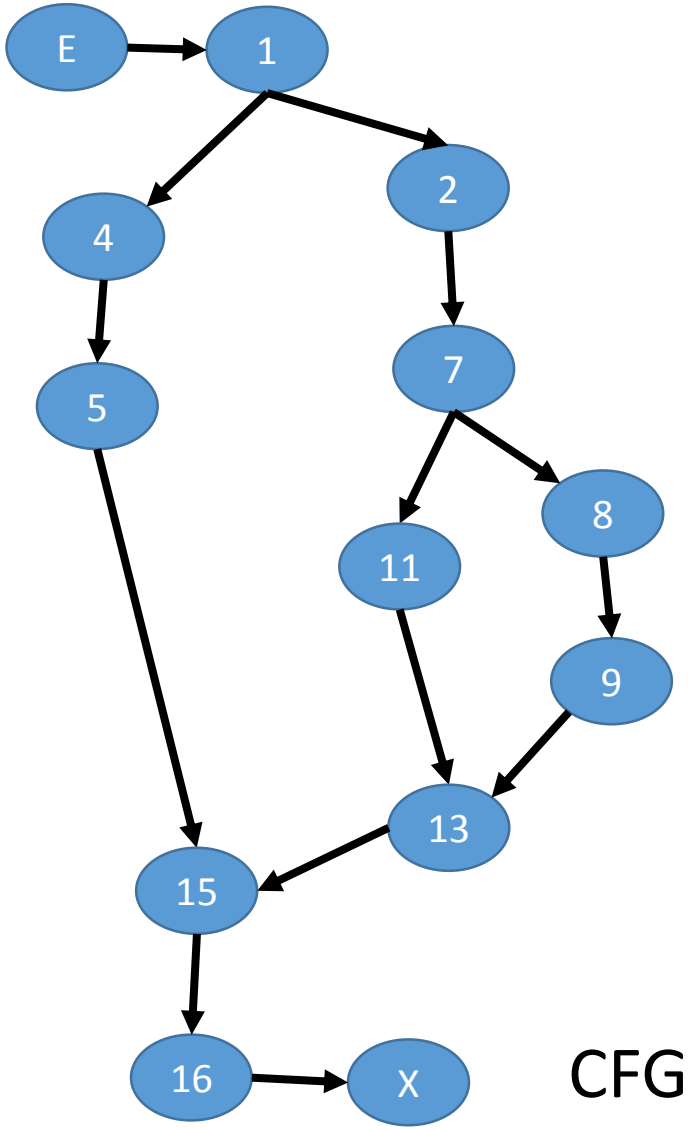
Unstructured programs

- GOTO statements define only unconditional control flow
  - No node can be control dependent on a GOTO statement
  - No node can be data dependent on a GOTO statement
  - GOTO statements will never get included in a slice
- One can produce goto less programs and then slice
  - The new program can be quite different textually

```
1.    if (x!=0)
2.      goto L1
3.    endif
4.    x = 10;
5.    goto L2
6.  L1:
7.    if (y > 0)
8.      w = 10
9.      goto L3
10.   endif
11.   w = 20
12. L3:
13.   x = 20
14. L2:
15.   t=x
16.   w = t
```

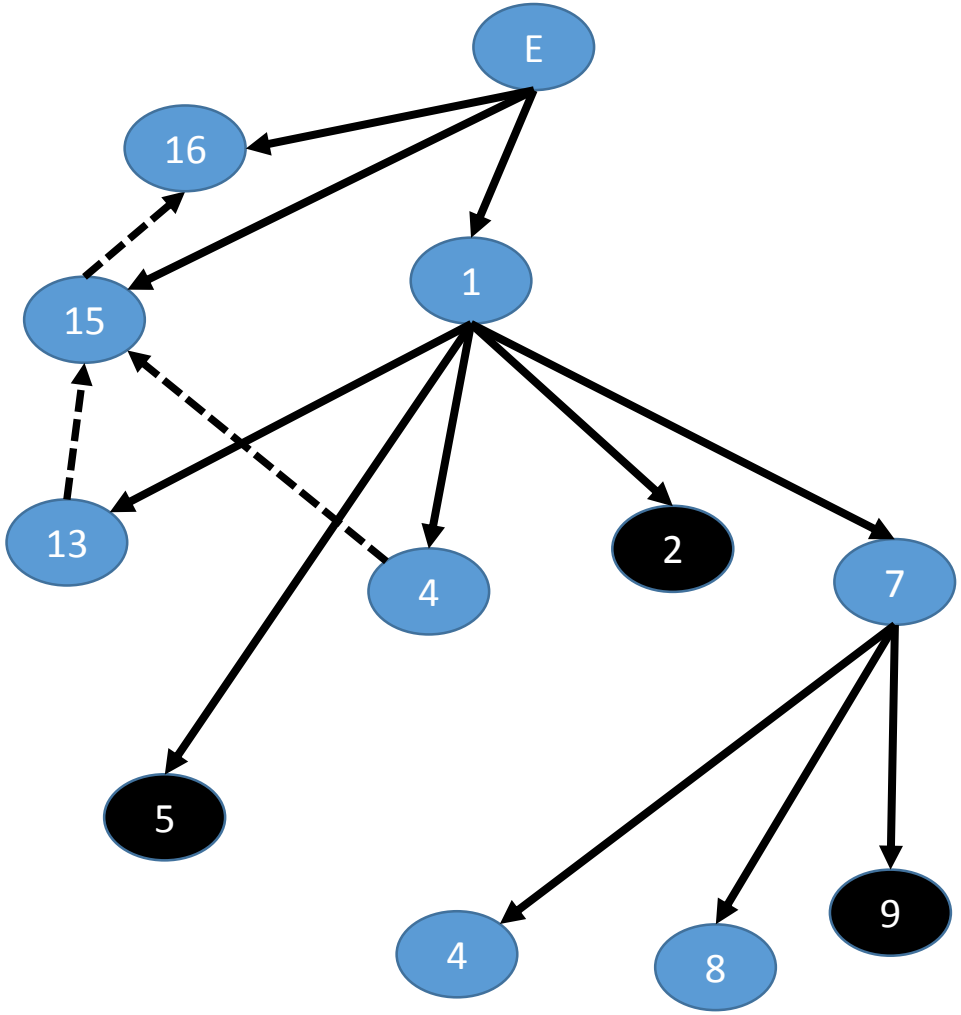
Many approaches have been suggested with varying degree of precision

Unstructured programs



CFG

```
1.    if (x!=0)
2.      goto L1
3.    endif
4.    x = 10;
5.    goto L2
6.  L1:
7.    if (y > 0)
8.      w = 10
9.      goto L3
10.   endif
11.   w = 20
12. L3:
13.   x = 20
14. L2:
15.   t=x
16.   w = t
```



PDG

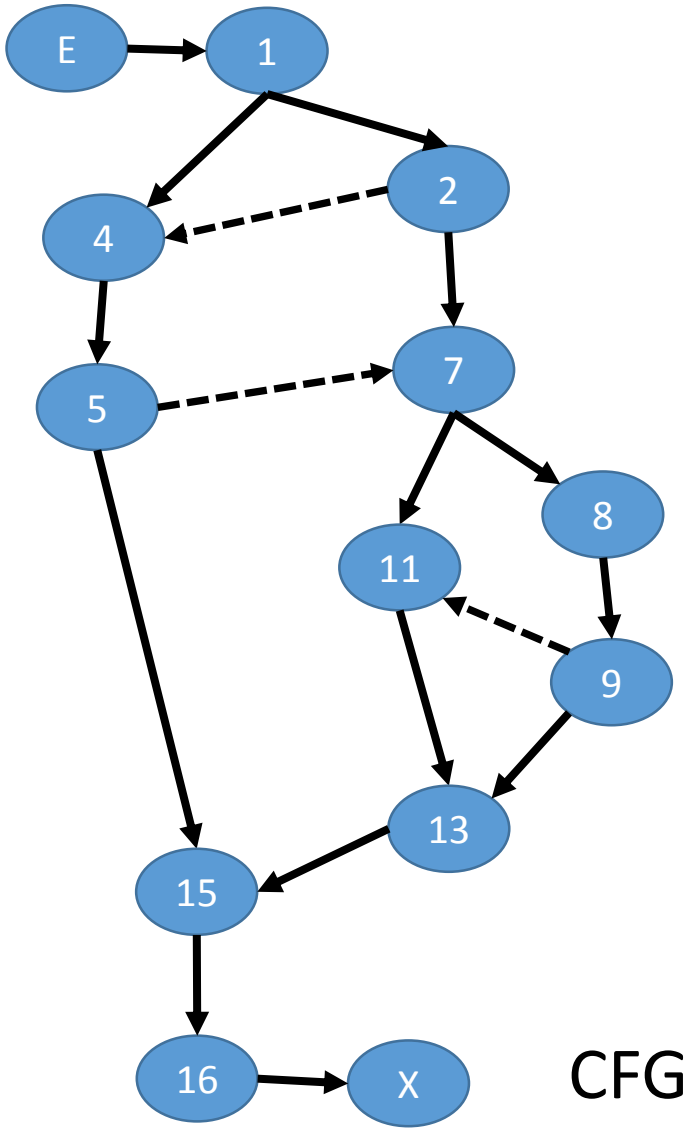
Slicing criterion : <16,{t}>

Unstructured programs

- In CFG, add auxiliary flow from GOTO node to textually successor statement
- Consider the auxiliary edges for control dependence only
- Perform slicing using the resulting PDG in usual manner
- Add the labels for GOTOs included in slice

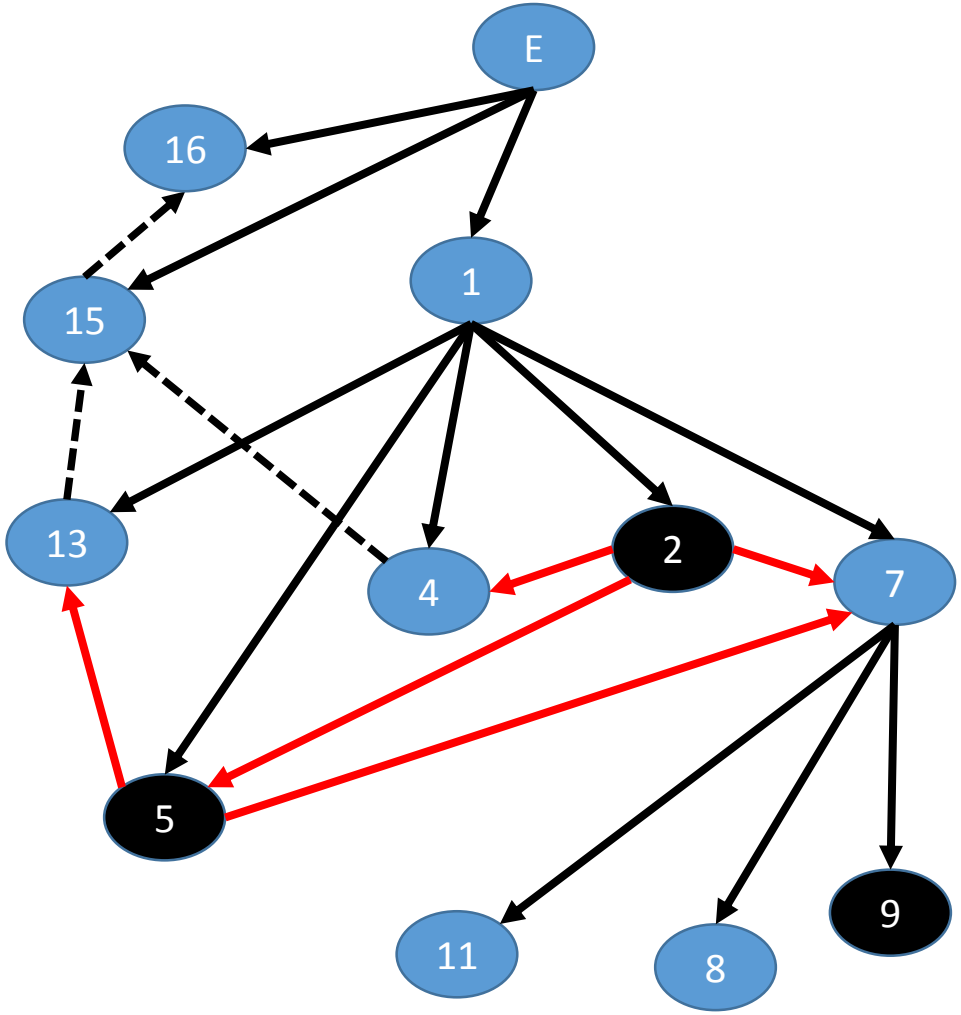
```
1.     if (x!=0)
2.       goto L1
3.     endif
4.     x = 10;
5.     goto L2
6.   L1:
7.     if (y > 0)
8.       w = 10
9.       goto L3
10.    endif
11.    w = 20
12.  L3:
13.    x = 20
14.  L2:
15.    t=x
16.    w = t
```

Unstructured programs



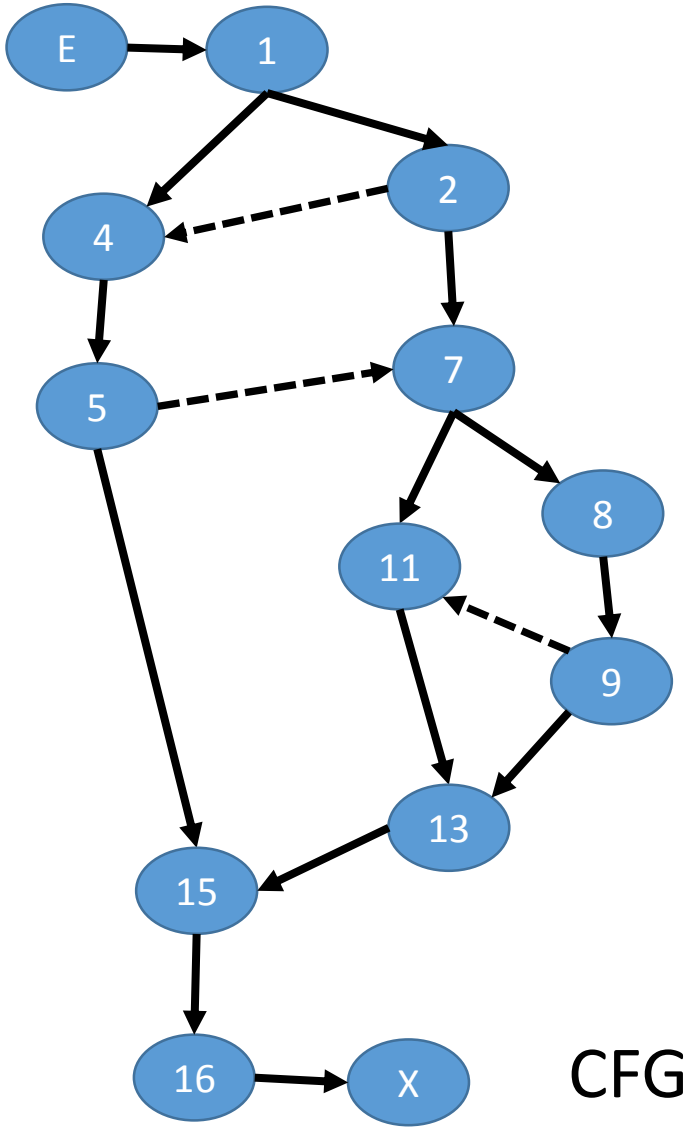
CFG

```
1.    if (x!=0)
2.      goto L1
3.    endif
4.    x = 10;
5.    goto L2
6.  L1:
7.    if (y > 0)
8.      w = 10
9.      goto L3
10.   endif
11.   w = 20
12. L3:
13.   x = 20
14. L2:
15.   t=x
16.   w = t
```



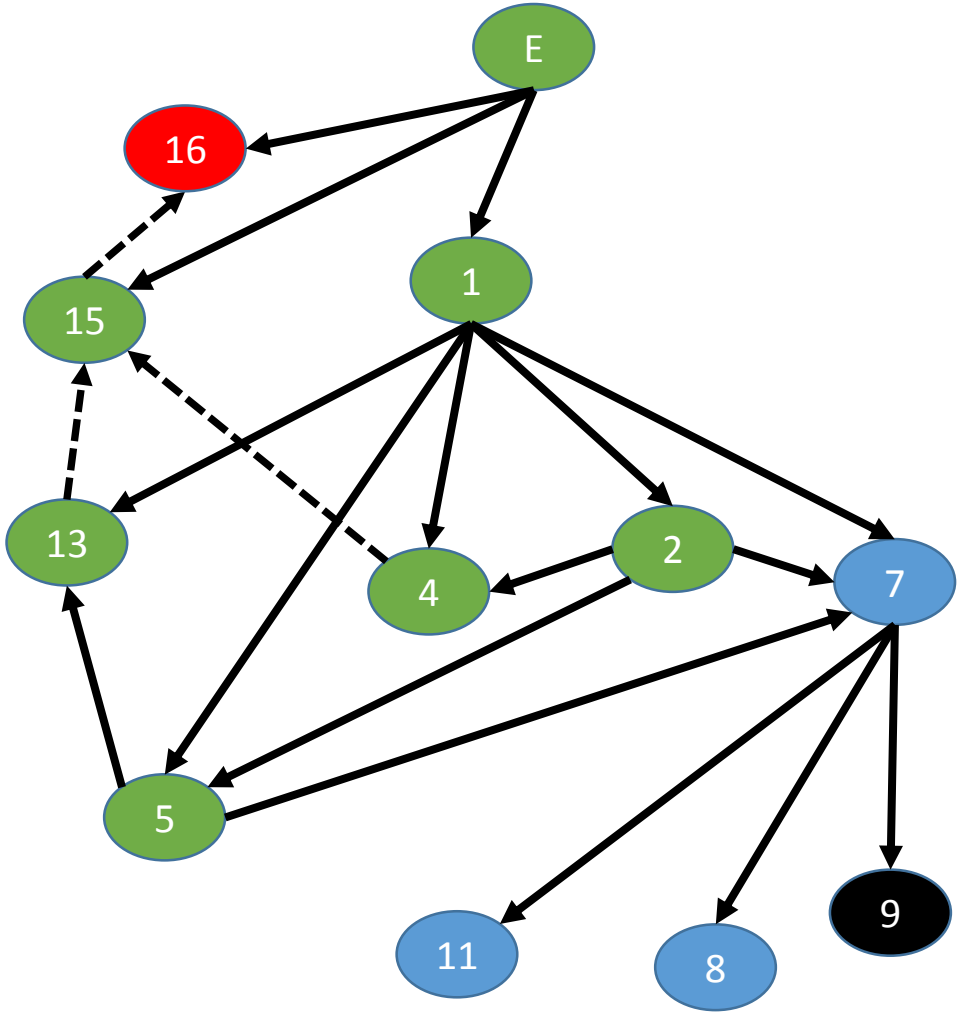
PDG

Unstructured programs



CFG

```
1.    if (x!=0)
2.      goto L1
3.    endif
4.    x = 10;
5.    goto L2
6.  L1:
7.    if (y > 0)
8.      w = 10
9.      goto L3
10.   endif
11.   w = 20
12. L3:
13.   x = 20
14. L2:
15.   t=x
16.   w = t
```



PDG

Slicing criterion : <16,{t}>

Arrays and pointers

- Need to answer questions:
  - Do a[e1] and a[e2] refer to same element ?
  - Can *p and *q refer to same variable ?
- In general, these questions are undecidable
- One way is to assume that a[e1] and a[e2] may refer to same element (results in quite imprecise slice)
- Use a points-to analysis to infer what does *p stand for
  - Steengard's or anderson's algorithm
  - Computing points-to using data flow analysis
- Precision of slice depends on precision of point-to information

```
a[k*i+c] = x;
…
y = a[h*i+d] ;

p = &t ;
q = &u;
if (x < 0)
   p = &y ;
*p = z   // can change t or y
if (x > 0)
  q = &y;
w = *q ; // can refer to y or u
```

Handling procedures

- Approaches discussed so far consider programs as single monolithic procedure
- What if program consist of multiple procedures ?
- How to address procedure call statement ?

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ;
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a;
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t);
25.     return z;
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- Approaches discussed so far consider programs as single monolithic procedure
- What if program consist of multiple procedures ?
- How to address procedure call statement ?

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ;
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a;
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t);
25.     return z;
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ;
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a;
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t);
25.     return z;  // {z}
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)
- For call at 24, we get a slicing criterion <18, a> (going down the call hierarchy)

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ;
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a; // {a}
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t);
25.     return z;  // {z}
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)
- For call at 24, we get a slicing criterion <18, a> (going down the call hierarchy)
- At 17, value of a and b needed

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ;
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a; // {a}
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t);
25.     return z;  // {z}
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)
- For call at 24, we get a slicing criterion <18, a> (going down the call hierarchy)
- At 17, value of a and b needed
- Before call to add at line 24 and 11, we get slicing criterion <24, {z,t}> and <11, {s,y}> (going up)

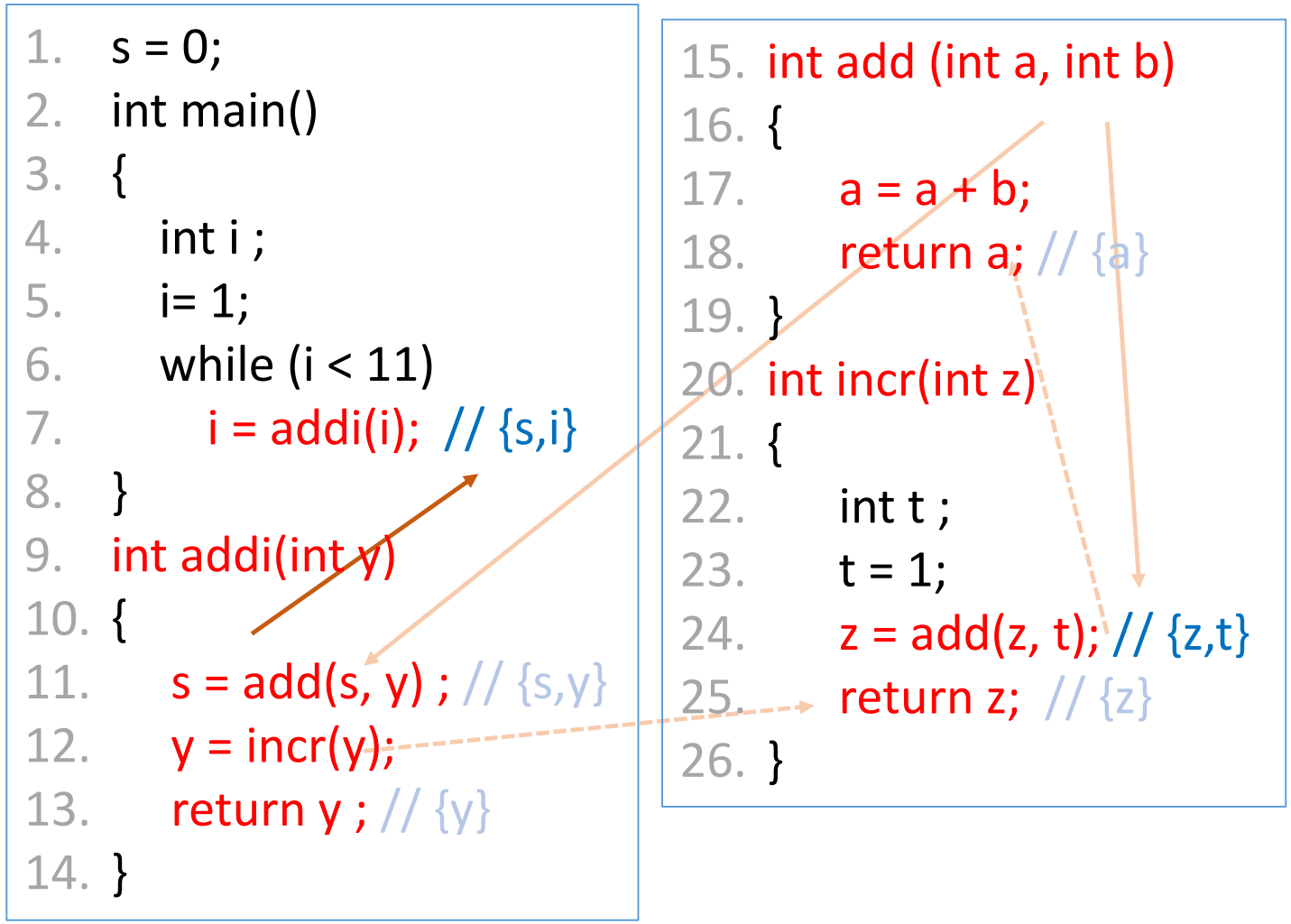
```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ; // {s,y}
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a; // {a}
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t); // {z,t}
25.     return z;  // {z}
26. }
```

Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)
- For call at 24, we get a slicing criterion <18, a> (going down the call hierarchy)
- At 17, value of a and b needed
- Before call to add at line 24 and 11, we get slicing criterion <24, {z,t}> and <11, {s,y}> (going up)
- So for call to addi at line 7, we get slicing criterion <7, {s,i}> (going up)



Slicing criterion : <13, {y}>

Handling procedures

- For call at 12, we get a slicing criterion <25, z> (going down the call hierarchy)
- For call at 24, we get a slicing criterion <18, a> (going down the call hierarchy)
- At 17, value of a and b needed
- Before call to add at line 24 and 11, we get slicing criterion <24, {z,t}> and <11, {s,y}> (going up)
- So for call to addi at line 7, we get slicing criterion <7, {s,i}> (going up)

Oops !! Everything in slice

```
1.  s = 0;
2.  int main()
3.  {
4.      int i ;
5.      i= 1;
6.      while (i < 11)
7.          i = addi(i);  // {s,i}
8.  }
9.  int addi(int y)
10. {
11.     s = add(s, y) ; // {s,y}
12.     y = incr(y);
13.     return y ; // {y}
14. }
```

```
15. int add (int a, int b)
16. {
17.     a = a + b;
18.     return a; // {a}
19. }
20. int incr(int z)
21. {
22.     int t ;
23.     t = 1;
24.     z = add(z, t); // {z,t}
25.     return z;  // {z}
26. }
```

Slicing criterion : <13, {y}>

Interprocedural slicing : issues

- Multiple slicing of same procedure
- The calling context
- Parameter passing

Interprocedural slicing as a graph reachability problem

System dependence graph (SDG) by S. Horwitz et al. (1990)

- A PDG constructed for every procedure
- Interconnection made from call site to entry point of called procedure
- Procedure entry point is taken to be control dependent on call site
- Additional edges added to depict data flow of actual to formal and return value
- Transitive dependence inferred to create additional dependence across the call between input-output

Slicing is done as finding reachability in the integrated graph

- While slicing a procedure do not go down a call immediately, just record the slicing criteria for called procedure
- After slicing a procedure go up to call places only if current procedure is being sliced in up-traversal

Dynamic slicing

- Static slice is set of statements which can influence the values identified by slicing criterion in any execution of the program

- In debugging, one examines possible causes of a bug observed in a given (single) execution of the program
  - Static slice may throw a much larger number of statements than one has to look at in debugging process

- Dynamic slice is set of statements which influence the values, identified by slicing criterion, in a given execution (identified by given input) of the program
  - Introduced by Korel & Laski (1988)

Example

```
1.  read(x)
2.  if (x < 0)
3.     y = f1(x)
4.     z = g1(x)
5.  else
6.     if (x=0)
7.        y = f2(x)
8.        z = g2(x)
9.     else
10.       y = f3(x)
11.       z = g3(x)
12.    endif
13. endif
14. write(y)
15. write(z)
```

```
1.  read(x)
2.  if (x < 0)
3.     y = f1(x)
4.     z = g1(x)
5.  else
6.     if (x=0)
7.        y = f2(x)
8.        z = g2(x)
9.     else
10.       y = f3(x)
11.       z = g3(x)
12.    endif
13. endif
14. write(y)
15. write(z)
```

Static slice

```
1.  read(x)
2.  if (x < 0)
3.     y = f1(x)
4.     z = g1(x)
5.  else
6.     if (x=0)
7.        y = f2(x)
8.        z = g2(x)
9.     else
10.       y = f3(x)
11.       z = g3(x)
12.    endif
13. endif
14. write(y)
15. write(z)
```

Dynamic slice for x = -1

```
1.  read(x)
2.  if (x < 0)
3.     y = f1(x)
4.     z = g1(x)
5.  else
6.     if (x=0)
7.        y = f2(x)
8.        z = g2(x)
9.     else
10.       y = f3(x)
11.       z = g3(x)
12.    endif
13. endif
14. write(y)
15. write(z)
```

Dynamic slice for x = 0

Dynamic slice definition – Korel & Laski

- A ***trajectory / execution history*** is sequence of statements executed for a given input
  - For n=1 , trajectory is 1,2,3,4,5,6,7,8,5,10,11

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

1^1 read(n)
2^1 z = 0
3^1 y=0;
4^1 i=1;
5^1 i <= n
6^1 z = f1(z,y)
7^1 y = f2(y)
8^1 i = i+1
5^2 i <= n
10^1 write(z)
11^1 write(y)

Dynamic slice definition – Korel & Laski

- A ***trajectory / execution history*** is sequence of statements executed for a given input
  - For n=1 , trajectory is 1,2,3,4,5,6,7,8,5,10
- Slicing criterion : <x, I^q, V>, e.g. <1, 10^1, {z}> with respect to the trajectory produced
  - Let T be trajectory and T_f be the prefix (front) of T till I^q

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

1^1	**read(n)**
2^1	**z = 0**
3^1	**y=0;**
4^1	**i=1;**
5^1	**i <= n**
6^1	**z = f1(z,y)**
7^1	**y = f2(y)**
8^1	**i = i+1**
5^2	**i <= n**
10^1	**write(z)**
11^1	write(y)

Dynamic slice definition – Korel & Laski

- A ***trajectory / execution history*** is sequence of statements executed for a given input
  - For n=1 , trajectory is 1,2,3,4,5,6,7,8,5,10
- Slicing criterion : <x, I^q, V>, e.g. <1, 10^1, {z}> with respect to the trajectory produced
  - Let T be the trajectory and T_f be the prefix (front) of T till I^q
- Dynamic slice is a subprogram P' of P such that if P' produces trajectory T' on input x, then
  - There is a prefix (front) T'_f of T' which ends at I^q
  - T'_f can be produced from T_f by deleting all statements which do not belong to P'
  - $\forall v \in V$, value of v before execution of I^q in T equals the value of v before execution of I^q in T'

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

1^1	**read(n)**
2^1	**z = 0**
3^1	**y=0;**
4^1	**i=1;**
5^1	**i <= n**
6^1	**z = f1(z,y)**
7^1	**y = f2(y)**
8^1	**i = i+1**
5^2	**i <= n**
10^1	**write(z)**
11^1	write(y)

Dynamic slice definition – Korel & Laski

- A ***trajectory / execution history*** is sequence of statements executed for a given input
  - For n=1 , trajectory is 1,2,3,4,5,6,7,8,5,10
- Slicing criterion : <x, I^q, V>, e.g. <1, 10^1, {z}>
  - With respect to the trajectory T produced by P on input x. Let T_f be the prefix of T till I^q
- Dynamic slice is a subprogram P' of P such that if P' produces trajectory T' on input x, then
  - There is a prefix T'_f of T' which ends at I^q
  - T'_f can be produced from T_f by deleting all statements which do not belong to P'
  - $\forall v \in V$, value of v before execution of I^q in T equals the value of v before execution of I^q in T'

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

```
1^1   read(n)
2^1    z = 0
3^1    y=0;
4^1    i=1;
5^1    i <= n
6^1    z = f1(z,y)
7^1    y = f2(y)
8^1    i = i+1
5^2    i <= n
10^1   write(z)
11^1   write(y)
```

Dynamic slice computation – Korel & Laski

- Like static slice, there is notion of statement-minimal dynamic slice and computing the same is undecidable in general

- Can we take the trajectory as a program and slice it as a static slice?

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

```
1^1   read(n)
2^1    z = 0
3^1    y=0;
4^1    i=1;
5^1    i <= n
6^1    z = f1(z,y)
7^1    y = f2(y)
8^1    i = i+1
5^2    i <= n
10^1   write(z)
11^1   write(y)
```

Dynamic slice computation – Korel & Laski

- Like static slice, there is notion of statement-minimal dynamic slice and computing the same is undecidable in general
- Can we take the trajectory as a program and slice it as a static slice?
  - No, something more needed

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

1^1 read(n)
2^1 z = 0
3^1 y=0;
4^1 i=1;
5^1 i <= n
6^1 z = f1(z,y)
7^1 y = f2(y)
8^1 i = i+1
5^2 i <= n
10^1 write(z)
11^1 write(y)

Dynamic slice computation – Korel & Laski

- Like static slice, there is notion of statement-minimal dynamic slice and computing the same is undecidable in general
- Can we take the trajectory as a program and slice it as a static slice?
  - No, something more needed
- Dynamic slice computation from trajectory
  - Data dependence can be taken as is
  - Extend the notion of control dependence to test instructions
    - So test at line 5^1 controls $6^1, 7^1, 8^1, 5^2$
  - If one occurrence of an instruction is included then all its occurrences in T_f to be included

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

```
1^1   read(n)
2^1   z = 0
3^1   y=0;
4^1   i=1;
5^1   i <= n
6^1   z = f1(z,y)
7^1   y = f2(y)
8^1   i = i+1
5^2   i <= n
10^1  write(z)
11^1  write(y)
```

Dynamic slice computation – Korel & Laski

- Like static slice, there is notion of statement-minimal dynamic slice and computing the same is undecidable in general
- Can we take the trajectory as a program and slice it as a static slice?
  - No, something more needed
- Dynamic slice computation from trajectory
  - Data dependence can be taken as is
  - Extend the notion of control dependence to test instructions
    - So test at line 5^1 controls $6^1, 7^1, 8^1, 5^2$
  - If one occurrence of an instruction is included then all its occurrences in T_f to be included

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

```
1^1   read(n)
2^1   z = 0
3^1   y=0;
4^1   i=1;
5^1   i <= n
6^1   z = f1(z,y)
7^1   y = f2(y)
8^1   i = i+1
5^2   i <= n
10^1  write(z)
11^1  write(y)
```

Dynamic slice computation – Korel & Laski

- Like static slice, there is notion of statement-minimal dynamic slice and computing the same is undecidable in general
- Can we take the trajectory as a program and slice it as a static slice?
  - No, something more needed
- Dynamic slice computation from trajectory
  - Data dependence can be taken as is
  - Extend the notion of control dependence to test instructions
    - So test at line 5^1 controls $6^1, 7^1, 8^1, 5^2$
  - If one occurrence of an instruction is included then all its occurrences to be included

```
1.  read(n)
2.  z = 0
3.  y=0;
4.  i=1;
5.  while (i <= n)
6.      z = f1(z,y)
7.      y = f2(y)
8.      i = i+1 ;
9.  endwhile
10. write(z)
11. write(y)
```

```
1^1   read(n)
2^1   z = 0
3^1   y=0;
4^1   i=1;
5^1   i <= n
6^1   z = f1(z,y)
7^1   y = f2(y)
8^1   i = i+1
5^2   i <= n
10^1  write(z)
11^1  write(y)
```

Dynamic slice computing algorithm

C : <x, I^q, V> : Slicing criterion, **B(n)** : Test condition which controls statement at n
LD(n) : statements which give last definition to variables used at n
LDC : statements which give last definition to variables in V at slicing criterion point

$$A^0 = LDC \cup B(I^q)$$

$$S^0 = A^0$$

$$S^{i+1} = S^i + A^{i+1}$$

$$A^{i+1} = \{ X^p \mid X^p \notin S^i \; and \; X^p \in T_f \; and \; \exists \, Y^t \in A^i . \; \mathrm{X^p} \in LD(Y^t) or \; X^p = B(Y^t) \; or \; (X = Y)\}$$

Slice is projection of S^k such that A^{k+1} is {}

Dynamic slice computing using PDG : Agrawal & Horgan (1990)

- Identify a subgraph of PDG using the trajectory
  - Nodes which got traversed
  - Edges which got traversed

- Restrict the traversal of PDG (for slicing) to the subgraph

- Restriction through edges traversed gets the same slice as one by Korel & Laski algorithm

Dynamic slice computing using PDG : Agrawal & Horgan (1990)

- Restriction by nodes or edges can produce larger dynamic slices

- Create the PDG for whole trajectory and use it for slicing
  - Dynamic dependence graph
  - Can have unbounded size

- Create a reduced dynamic dependence graph (merge the nodes with same transitive dependencies)

Static slice vs dynamic slice : Pros & Cons

- Static slice
  - Pros
    - Easy to compute
    - Preserves behavior for all runs
    - Good for regression testing, sematic differences, program comprehension
  - Cons
    - Too large to be of use for debugging
    - Imprecision due to arrays, pointers etc

- Dynamic slice
  - Pros
    - No imprecision due to arrays, pointers etc
    - Small size and good for debugging
  - Cons
    - Computing time and space complexity is high (trajectories could be huge)
    - Slice is useful only for debugging and not for what-if kind of queries

Some other variants of program slicing

- Chopping : Intersection of forward and backward slices [Daniel Jackson et al., 1994]
  - Statements which impact a slicing criterion <s1, X1> and get impacted by a slicing criterion <s2, X2>
- Conditioned : a generalised slicing [Genardo Canfora et al., 1998]
  - Set of statements which preserve the behavior of program with respect to a slicing criterion <s, X> along a given set of execution paths (specified by a first order logic formula over input variables)
- Thin slice : a non-executable static slice ignoring control dependence [M. Sridharan et al., PLDI 2007]
  - Resulting slice is small
  - Useful in debugging in neighbourhood
- Value slice : an executable but relaxed static slice [S. Kumar et al., TACAS 2015]
  - Values are same when observed on prefixes of same size of two trajectories
  - Resulting slices are smaller than usual backward slice
  - Useful in scaling up of model checking

To summarise …

- Program slicing
  - A subprogram which preserves the behavior of original program with respect to a slicing criterion <s, X>
- Static backward slice computation
  - Using data flow equations
  - Using PDG
  - Issues faced
- Variants of program slicing:
  - static/dynamic, forward/backward, executable/non-executable
- Dynamic slicing and its computation
- Some other variants of program slicing

References

- M. Weiser., *Program Slicing,* Proc. of the Fifth International Conference on Software Engineering, pages 439-449, May 1981
- D. Binkley and K. Gallagher., *Program Slicing,* Proc. of In Advances in Computers, Volume 43, 1996
- K. Ottenstein and L Ottenstein., The program dependence graph in a software development environment, ACM SIGPLAN Notices, Vol 19, May 1984
- J. Choi and J. Ferrante, Static slicing in presence of GOTO statements, ACM TOPLAS, May 1994
- H. Agrawal, On slicing programs with jump statements, Proceedings of PLDI (1994)
- S. Horwitz, T. Reps and D. Binkley., Interprocedural slicing using dependence graphs, ACM Transactions on programming languages and systems, Vol 12, No. 1, January 1990
- B. Korel and J. Laski., Dynamic program slicing, Information processing letters, 29:155-163, October 1988
- H. Agrawal and J. Horgan, Dynamic program slicing, ACM SIGPLAN Notices, Vol 25, No. 6, June 1990
- M. Sridharan, S.J. Fink, R. Bodik, Thin slicing, Proceedings of PLDI(2007)
- S. Kumar, A. Sanyal and Uday Khedker, Value slice: A new slicing concept for scalable property checking, Proceedings of TACAS(2015)