Program Analysis : Slicing

Submitted By

Nikhitha V Snehal Basavnathe Vipin S

CS16M047 CS16M048 CS12B029

Instructor Name

Dr. Rupesh Nasre

Scribe Week:17^{th} April - 21^{st} April, 2017

1 Computing Relevant Variables

1.1 Introduction

This section focuses more formally into how to select relevant variables and statements from a given slicing criteria. A slicing criteria, C is denoted as $<i, V>$, where i stands for the line number and V stands for the set of variables we are interested to track from line i. Relevant variables of a slicing criteria are computed from its relevant statements. The relevant statements, in turn are computed from the relevant variables.

eg: Let's take a look at the following statements:

```
1.y = x;
2.a = b;
3.z = y;
```

Let the slicing criteria be $<3, y>$, which means we would like to figure out which all variables and statements are likely to affect the value of y in the line number 3.
It's easy to say line 2 does not affect the value of y in 3 and line 1 does, and x and y are the relevant variables by intuition.

How to arrive at that conclusion?
The notation $R_c^j(n)$ denotes the set of variables relevant at line number n, for slicing criteria $C <i, V>$ at level j.
Step 1:

- Start with the line given by slicing criteria <i, V>
- add all the variables in V to it′s set of relevant variables R_c^j(i)

Step 2:

- for all the lines n, which are predecessors of the last processed line, say m
- add all the variables v such that:
  - (a) v ∈ use(n) and def(n) ∩ R_C^0(m) ≠ ∅
  - (b) v ∉ def(n) and v ∈ R_C^0(m)

repeat step 2 until you find the relevant variables for all the program statements.

Solving the example:

$R^0_{<3,y>}$ (3) = {y} - - - - by Step 1
$R^0_{<3,y>}$ (2) = {y} - - - - by Step 2b
$R^0_{<3,y>}$ (1) = {x} - - - - by Step 2a

Question: Why should y be removed from the set of relevant variables in line 1?
Answer: Since the variable y is written to at line 1, all definitions of y before line 1, say line 0, will be overwritten by the definition at line 1. Values of y in the predecessors of line 1 will not affect the event of interest.

```
0.  y = w;
1.  y = x;
2.  a = b;
```

```
3.  z = y;
```
The possibility of a segmentation fault in line 3 surely can't be caused by the line 0.

Question: When the variable defined in a line be retained relevant in the line?
Answer:
Case1: If line 1 was,
```
y = y + x;
```
Both x and y are relevant variables for this line since both x and y are used in this line.

Case2: If a statement has two predecessors, one with a definition of y and one with no definition, then even after the definition of y in one of the predecessors, y will continue to be relevant through the other predecessor.

```
0.w = y;
1.If(*)
2.    y = x;
3.else
4.    a = b;
5.z = y;
```
here, y is a relevant variable in line 0.

Relevant statements:
All the statements which define a relevant variable of its successor is a relevant statement. The statement i in the slicing criteria $<i, V>$ is a relevant statement by default.
Notation : S_c^i = set of relevant statements at level i for slicing criteria C

1.2 Including Control Dependence

Question : How a program slice differs from a use-def chain?
Answer: A program slice includes control dependence into consideration whereas a use-def chain is concerned only about data flow.

Example:
```
1.if(c==0)
2.  y = x;
```
There is no use of relevant variables in the *if* statement. But still it affects the execution of the statement $y = x$, therefore is added to the current set of relevant branch instructions.

INFL(b) is the set of statements which directly affect execution of statement b.

Therefore in the example,
INFL(2) = 1
B_c^i - Set of relevant branch instructions at level i.
$B_c^i = \cup$ INFL(n) such that n $\in S_c^i$

Question: What is the difference between direct and indirect influence?
Answer: A statement a is directly influenced by a branch statement b if there is no other branch statement controlling the flow from b to a.

1.3 Computing a static slice

Relevance at level n is defined as follows:
Next set of relevant variables = current set of relevant variables ∪ relevant variables in current set of relevant branch statements

for all i >= 0
R_c^{i+1}(n) = R_c^i(n) ∪ $R^0_{BC(b)}$(n) for b ∈ B_c^i where BC(b) is a branch statement criterion defined as <b, use(b)>
B_c^{i+1} = ∪ INFL(n) for n ∈ S_c^{i+1}
S_c^{i+1} = all statements n such that def(n) ∩ R_c^{i+1}(n+1) ≠ ∅ or n ∈ B_c^i

Question: Why is it always $R^0_{BC(b)}$(n) rather than $R^i_{BC(b)}$(n)?
Answer: Set of variables used in the newly included set of branch statements is included in by $R^0_{BC(b)}$(n)to the next level of relevant variables. No matter what the level number i is, it's always $R^0_{BC(b)}$(n) to add the variables used by the newly added set of branch statements.

Classwork:

```
1.void main() {
2.int y, z, ii, n;
3.int *a;
4.scanf("%d", &n);
5.a = (int*)malloc(n);
6.for(ii = 0;
7.      ii < n; ){
8.      scanf("%d", a + ii);
9.      ++ii;
10.}
11.y = 0;
12.z = 1;
13.for(ii = 0;
14.      ii < n;
15.      ++ii) {
16.      y += a[ii];
17.      z *= y;
18.}
19.printf("%d\n", z);
20.}
```

Slicing criteria : <19, z>

Level 0 :
$R^0_{<19,z>}$ (19) = {z}
$R^0_{<19,z>}$ (14) = {z}
$R^0_{<19,z>}$ (15) = {z}
$R^0_{<19,z>}$ (17) = {z,y}
$R^0_{<19,z>}$ (16) = {z,y,a,ii}
$R^0_{<19,z>}$ (13) = {z,y,a}
$R^0_{<19,z>}$(12) = {y,a}
$R^0_{<19,z>}$ (11) = {a}
$R^0_{<19,z>}$ (7) = {a}

$R^0_{<19,z>}$ (9) = {a}
$R^0_{<19,z>}$ (8) = {a}
$R^0_{<19,z>}$ (6) = {}
$R^0_{<19,z>}$ (5) = {}
$R^0_{<19,z>}$ (4) = {}
$R^0_{<19,z>}$ (3) = {}
$R^0_{<19,z>}$ (2) = {}

$S^0_{<19,z>}$ = {19,17,16,15,13,12,11,8}
$B^0_{<19,z>}$ = {14,7}

Level 1:
$R^1_{<19,z>}$ (19) = {z} $\cup$ $R^0_{<14,\{ii,n\}} > \cup R^0_{<7,\{ii,n\}>}$
= {z} $\cup$ {} $\cup$ {}
= {z}
$R^1_{<19,z>}$(14) = {z} $\cup$ $R^0_{<14,ii,n>} \cup R^0_{<7,ii,n>}$
= {z} $\cup$ {ii,n} $\cup$ {}
= {z,ii,n}
$R^1_{<19,z>}$ (15) = {z,ii,n}
$R^1_{<19,z>}$ (17) = {z,y,ii,n}
$R^1_{<19,z>}$ (16) = {z,y,a,ii,n}
$R^1_{<19,z>}$ (13) = {z,y,a,n}

$R^1_{<19,z>}$ (12) = {y,a,n}
$R^1_{<19,z>}$ (11) = {a,n}
$R^1_{<19,z>}$ (7) = {a,n} $\cup$ $R^0_{<14,\{ii,n\}>} \cup R^0_{<7,\{ii,n\}}>$
= {a,ii,n}
$R^1_{<19,z>}$ (9) = {a,n,ii}
$R^1_{<19,z>}$ (8) = {n,ii}
$R^1_{<19,z>}$ (6) = {n}
$R^1_{<19,z>}$ (5) = {n}
$R^1_{<19,z>}$ (4) = {}
$R^1_{<19,z>}$ (3) = {}
$R^1_{<19,z>}$ (2) = {}

$S^1_{<19,z>}$ = {19,18,17,16,15,14,13,12,11,10,9,8,7,6,4}
$B^1_{<19,z>}$ = {14,7}

1.4 Slicing as a DFA

Slicing may not always lead to the minimal set of statements. The following example illustrates when it fails to include only the relevant statements.

```
1.a = 1;
2.while(f(k)) {
3.    if(g(c)) {
4.         b = a;
5.         x = 2;
6.    } else {
7.         c = b;
8.         1 = 3;
```

```
    }
9.        k = k + 1;
}
10.z = x+y;
```

In the first level of indirection statements 10, 8 and 5 are included in the set of relevant statements.And set of relevant variables is *{x, y}*

But execution of 8 and 5 are dependent on 3, which will add *c* in the set of relevant variables.Now statement 7 and thereby 4 is included in the relevant statements. And relevant variables at line 3 = *{c, a}* execution of *if* is dependent on the *while* condition therefore *while* is also added to the list of relevant statements.eventually all the statements are added to the set of relevant statements including *a=1.*

Question: Are all the statements actually relevant?

Answer: Notice that the only place where *c* is getting defined is in the else part. So once *g(c)* is true, the if part is executed and c is never changing.The value of *g(c)* continues to be true always. Once statement 4 is executed, statement 7 is never executed and the data dependency from 4 to 7 does not exist.

So, adding 4^{th} and 1^{st} statements to the list of relevant statements is not actually needed.

2 Dynamic Slicing

Dynamic slice is a slice that preserves the program's behaviour for a specific input rather than that of set of all input for which program terminates. Slice is dynamic if it is done at execution time or compile time relying on specific test cases.
Slicing criteria: <statement instance, variable of interest, input given to program>
eg: Consider the following piece of code

```
1.  read(n);
2.  i = 1;
3.  while(i <= n) {
4.     if(i % 2 == 0)
5.        x = 7;
6.     else
7.        x = 16;
8.  ++i;
   }
9.  write(x);
```

Dynamic slice for above code for slicing criteria $< 9^1$, x, n = 2 >

```
1.  read(n);
2.  i = 1;
3.  while(i <= n) {
4.     if(i % 2 == 0)
5.       x = 7;
6.     else
7.       ;
8.  ++i;
   }
9.  write(x);
```

In first iteration, x is assigned to 16. But in second iteration, x is assigned to 7. The else branch may be omitted from the dynamic slice because assignment of 16 to x in first iteration is killed by assignment of 7 in second iteration.

3 Static vs Dynamic Slicing

Static	Dynamic
• Performed at compile time • across all set of inputs • less precise • source code or IR • slicing criteria contains statements and variables	• Performed on run time trace • specific set of inputs • more precise • source code or trace • slicing criteria contains statement instance, variables and input

- Precision is more in dynamic slicing as a trade off for soundness. Static slicing is more like a safety property whereas dynamic slicing is liveness property.
- Dynamic slicing is more precise for specific set of inputs.
- Static slicing often contains statements which have no influence on values of variables of interest. Dynamic slice can be considered as a refinement of static one. By applying dynamic analysis it is easier to identify those statements in the static slice which do not have influence on variables of interest thereby reducing the searching space for the fault in program.

4 Computing Slices:

Slices can be computed using reachability in dependence graph.

- Program Dependence Graph (intra procedural)
- System Dependence Graph (inter procedural)

If we include inter procedural edges to dependence graph it becomes SDG.

Dependence Graph:

- Control Dependence
- Data Dependence

4.1 Control Dependence:

Is control dependence same as control flow graph?
Answer: No.
Consider two statements
S1: `x = 3;`
S2: `y = 5;`

There is flow from S1 to S2, but they are not control dependent. In general we assume that statements terminate, hence execution of one statement does not depend on the execution of other.

Class work: Find the control dependence graph for the following program

```
void main() {
   int sum = 0;
   int i = 0;
   while(i < 11) {
      sum += i;
      ++i;
   }
   printf("sum = %d\n", sum);
   printf("i = %d\n", i);
}
```

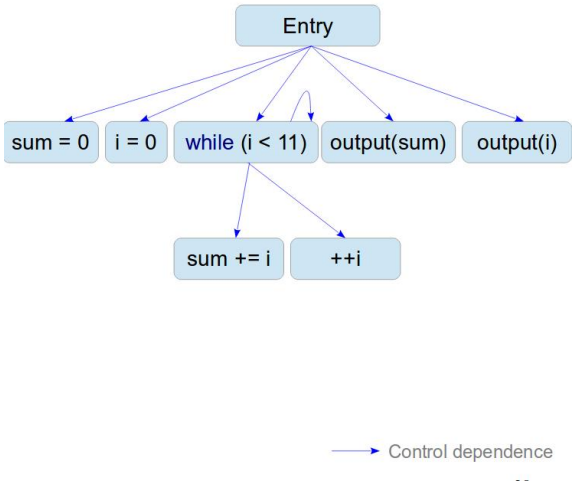

Figure 1: Control Dependency Graph

Question: Will entry have the control dependence on all the statements?
Answer: Control dependence is transitive. In above figure, there is path from entry to *sum += i* statement via *while* statement. Entry will have control dependence on all the statements.

Question: There are two *printf()* statements in program. Should those be control dependent on *while* or on *main*?
Answer: There is no control dependence between two *printf()* statements as execution of first *printf()* is not dependent on execution of second *printf()*. Execution of statements is guarded by control. These control statements can be of any type such as *for*, *while*, *if*, *if-else* or *goto*. As in *while* loop, *sum += i* and *++i* are guarded by condition $i < 11$. Hence there is an edge from *while* statement to these two statements. But there is no control dependence from *while* statement to any of *printf()* statements. So there is edge from entry to two *printf()* statements.

Question: If there is *if* statement after *while*, will that be control dependent on *while* statement?

Answer: The statements inside *while* loop are guarded by *while* condition so they are control dependent on *while*. But as *if* statement is outside *while* loop, it is always going to get evaluated irrespective of the number of times the *while* loop is executed. Hence there is no control dependence from *while* to *if* statement but there is control flow from *while* to *if*.

Question: Consider the following piece of code.

```
while(//some condition)
{
   sum += i;
   if(//some condition)
      x = 4;
}
if(x == 0)
   S4;
```

Which of the statements has control dependence on *while*?
Answer: Inside *while* loop, *sum += i* and *if* condition are present. Hence those are control dependent on *while*. Second *if* statement is outside *while* loop which is always going to get executed. There is definition of x in first *if* statement and use of x in second *if*, hence those two are data dependent but not control dependent. S4 is guarded by second *if* condition, hence S4 is control dependent on second *if*.

Note: Control flow is different from control dependence.

Class work: Draw data dependence graph for the same program.

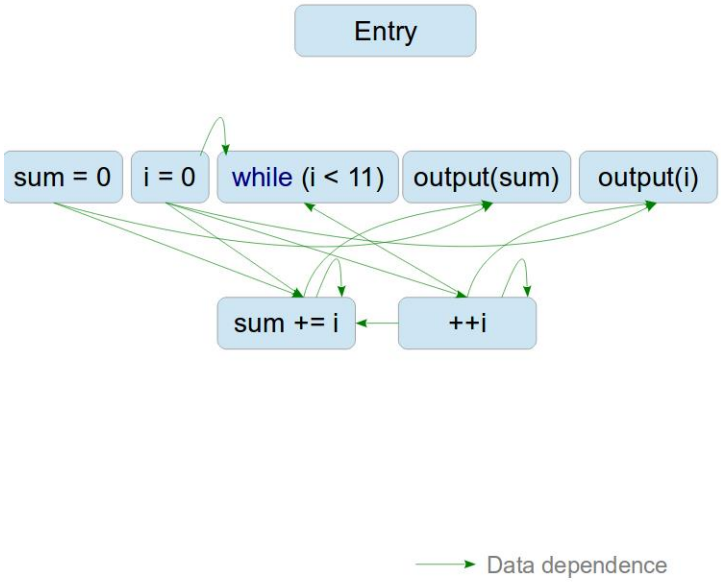

Figure 2: Data Dependency Graph

Question: Is data dependence transitive?
Answer: No.
Consider following example.

```
b = c;  // def of b
a = b;  // use of b
d = a;  // use of a
```

But there is no data dependence from `b = c` to `d = a`. Hence there should not be any edge from `b = c` to `d = a`. Only direct data dependence edges are added in the graph.

4.2 Program Dependence Graph(PDG):

PDG is combination of control dependence and data dependence graph. Control and data both are required to figure out the dependencies across different statements.

Following graph shows PDG for the same program.

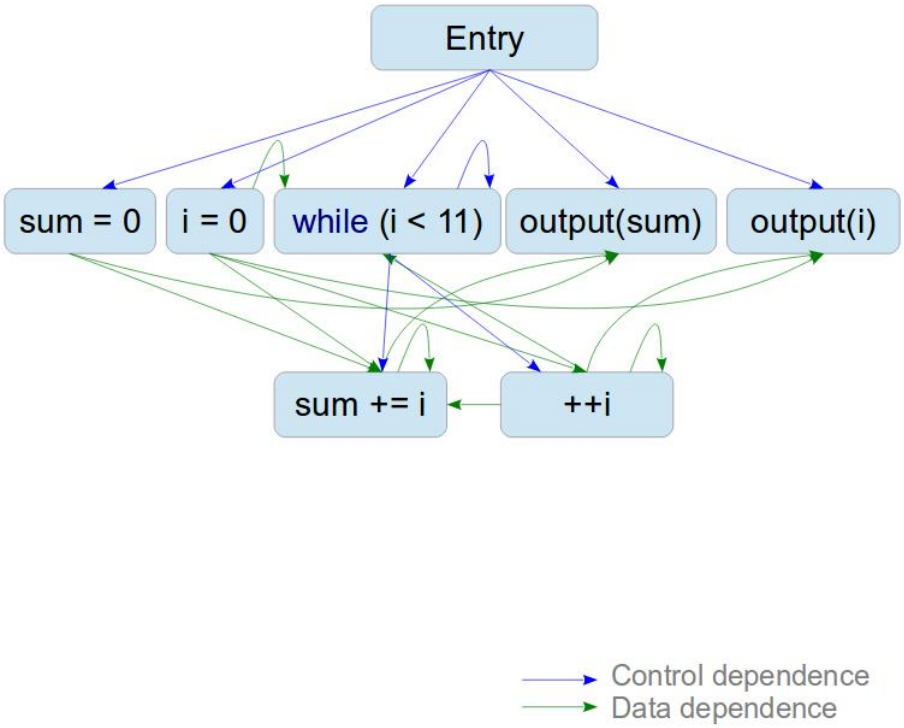

Figure 3: Program Dependency Graph

PDG helps in finding out the program slice. Back trace PDG from the slicing criteria and find the statements on which it is dependent. The statements obtained constitutes the slice for that slicing criteria.

5 Syntactically Valid Slices

There might be a requirement for the slices computed to be syntactically valid. Human involvement might be needed to automatically convert a particular slice to a functionality, since the set of statements identified might not be leading to valid slices always.

It is slightly easy in C due to the presence of ;, but even in C errors can arise if not dealt with care. In more free form languages errors can happen more frequently

The lines of code computed may not satisfy a syntactically valid slice, however if a few more statements are added, then together it can form a syntactically valid slice. This is driven by the grammar rules of the language, and typically enclosing of if or else conditions may be needed.

We are interested in errors which happen at the source level. The errors which happen at the IR level can be similarly handled.

There are two ways in which errors can propagate. First case will be illustrated with the case given here with <7,n> as slicing criteria.

```
1.read(n);
2.i = 1;
3.if (p == null)
4.   x = x + 1;
5.else
6.   n = n * 2;
7.write(n);
```

The result we get is the slice

```
if (p == null)

else
   n = n * 2;
write(n);
```

This is a syntactically invalid code leading to compile time error, but can be easily taken care of in languages like C with a semicolon.

The computed slice can also turn out to be syntactically valid, but semantically different from the original code. For example changing n in the previous example to x.

```
read(n);
i = 1;
if (p == null)
   x = x + 1;
else
   n = n * 2;
write(x);
```

The result we get is the slice

```
if (p == null)
   x = x + 1;
else

write(x);
```

In this statement if we use *<7,x>* as slicing criteria, the slice we get is syntactically valid in C, the semantics have changed, and the *write(x)* have become part of *if-else.*

Thus slicing can cause errors in the syntax and/or errors in the semantics of the program.

Class work: For the following program

```
1 main() {
2 int x = 0;
3 int n = 1;
4 int a[5] = {0, 2};
5 int i = 0;
6 if (n > 0) {
7    for (; i < n; ++i) {
8       a[i] = i * i;
9       if (a[i] < 100) {
10          x += a[i];
11       } else {
12          x = a[i];
13       }
14    }
15 }
16 printf("%d\n", x);
17 }
```

- draw control-dependence graph.
- draw data-dependence graph
- find forward slice for <4, a>.
- find backward slice for <16, x>

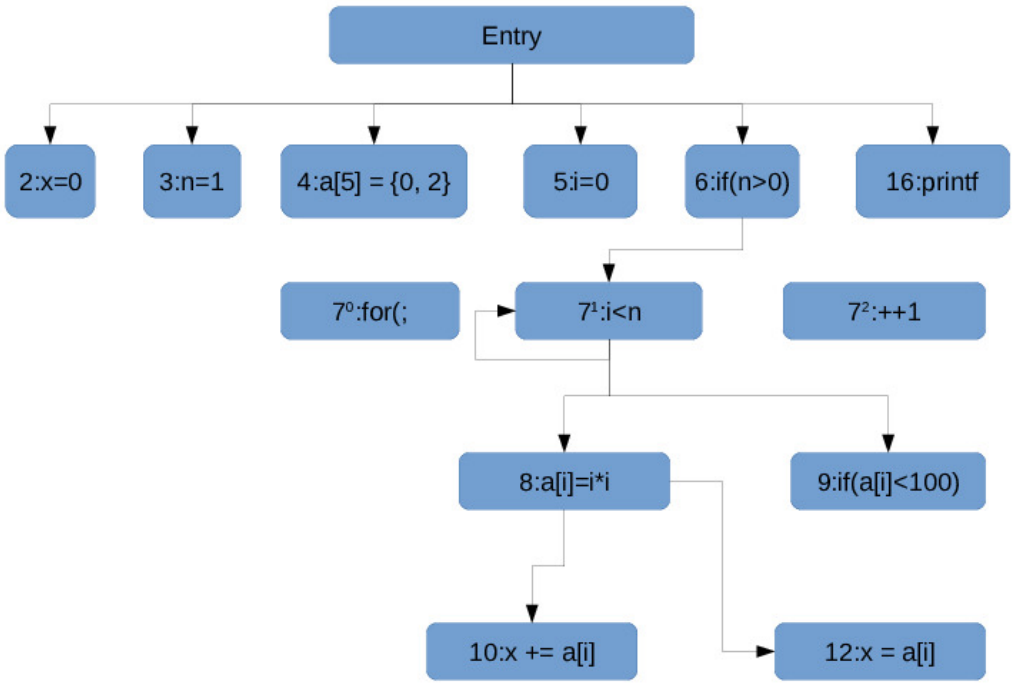

Figure 4: Control Dependency Graph

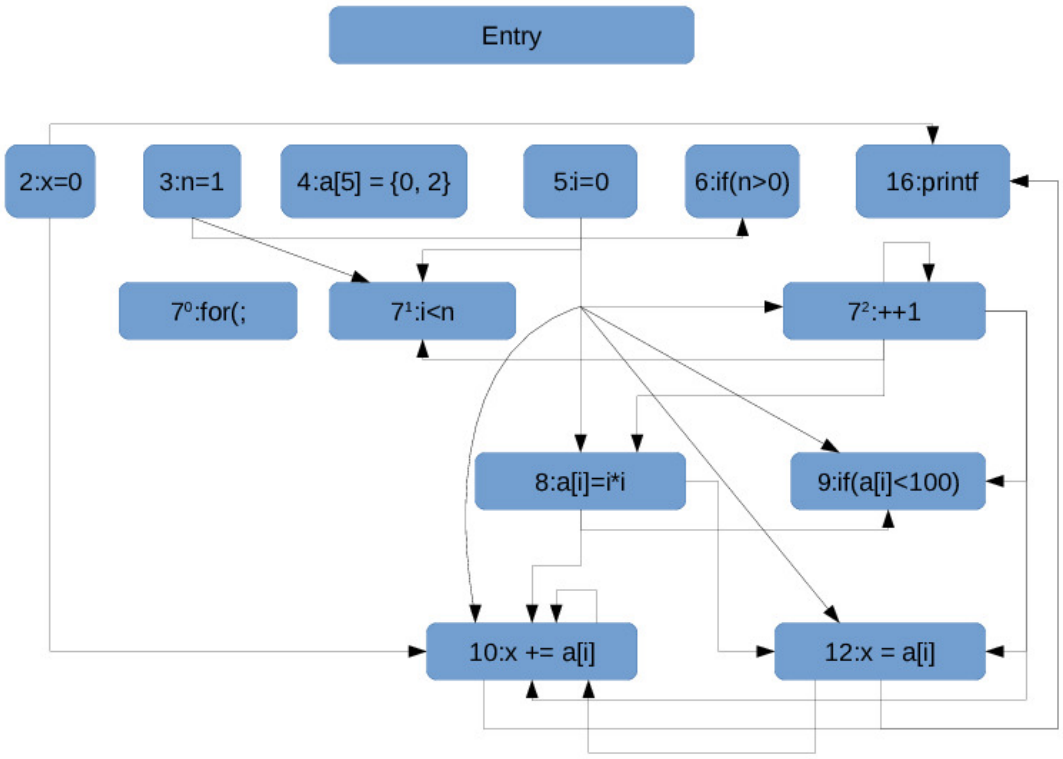

Figure 5: Data Dependency Graph

Note that in the Data Dependence graph no usage of *a[i]* is dependent on line 4. The analysis is field sensitive, and therefor finds out that the definitions of a[0] and a[1] in line 4 is overwritten by line 8. Analysis also figures out that the fields referred in line 8, 9, 10 and 12 are the same thereby adds no data dependency edge from 4 to 8, 9, 10 and 12. In field insensitive, may analysis, the field information is not tracked and it is not possible to figure out whether different accesses to array *a* are same or not. To kill *a[i]* definition at line 4, a must information is needed, which cannot be done in a field insensitive manner. In may analysis fields of *a* are indistinguishable, while in a must analysis each field of *a* is being tracked. Thus in field insensitive analysis we will not be able to kill a[i] and thus all the uses of a[i] will be data dependent on line 4.

Forward slice for <4, a>

Since there is no out-edge from <4, a> in Program dependency graph, the forward slice of <4, a> is only that line.

Thus the Forward slice of <4, a>

```
1.  main() {
4.  int a[5] = {0, 2};
```

```
}
```

Backward slice for <16, x>

Repeatedly apply the backward dependence algorithm above. First look at the data dependence on line 16, which are 10 and 12. Now add their control and data dependencies. Work backwards with this algorithm by recursively applying it on the two graphs.

It can be seen easily that the only line that is not a going to be part of the slice is line 4.

Thus the Backward slice for <16, x> is

```
1.  main() {
2.  int x = 0;
3.  int n = 1;
5.  int i = 0;
6.  if (n > 0) {
7.     for (; i < n; ++i) {
8.        a[i] = i * i;
9.        if (a[i] < 100) {
10.           x += a[i];
11.        } else {
12.           x = a[i];
13.        }
14.     }
15.  }
16.  printf("%d\n", x);
17.  }
```