Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

Indian Institute of Science, Bangalore

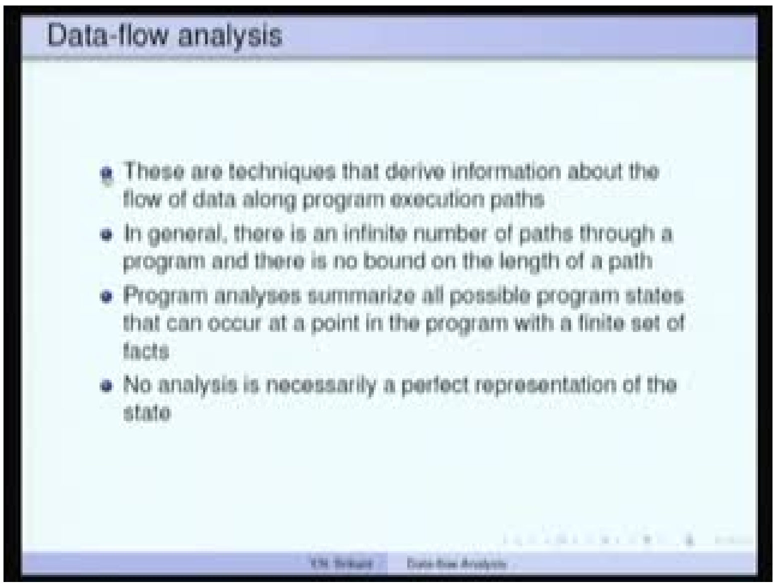
Module No. # 08

Lecture No. # 20

Data-flow Analysis – Part 2

Welcome to part-2 of the lecture on data-flow analysis. So, data-flow analysis derives information from the data along the program execution paths. In general, there is an infinite number of a path through a program and there is no bound on the length of a path.

(Refer Slide Time: 00:20)

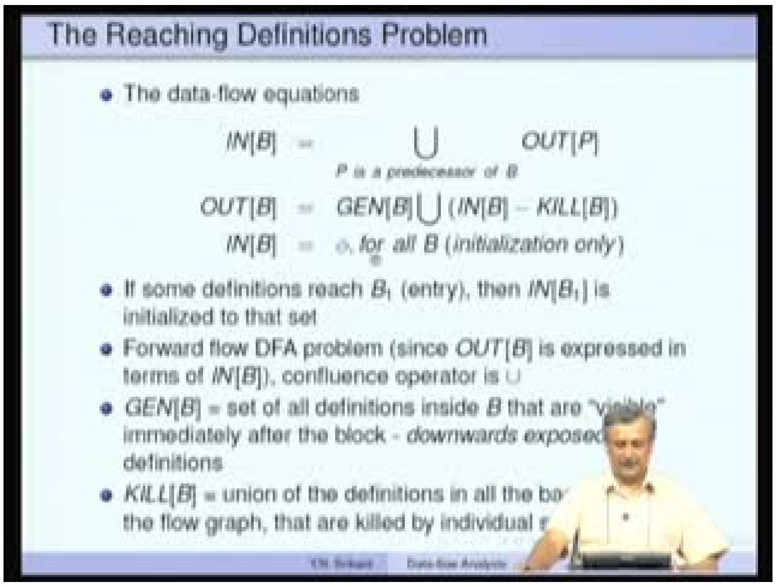


Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

So, program analysis actually summarizes all possible program states that can occur at a point in the program with a finite set of facts, and there is no analysis which is perfect representation of the state.

(Refer Slide Time: 01:03)



The Reaching Definitions Problem

- The data-flow equations

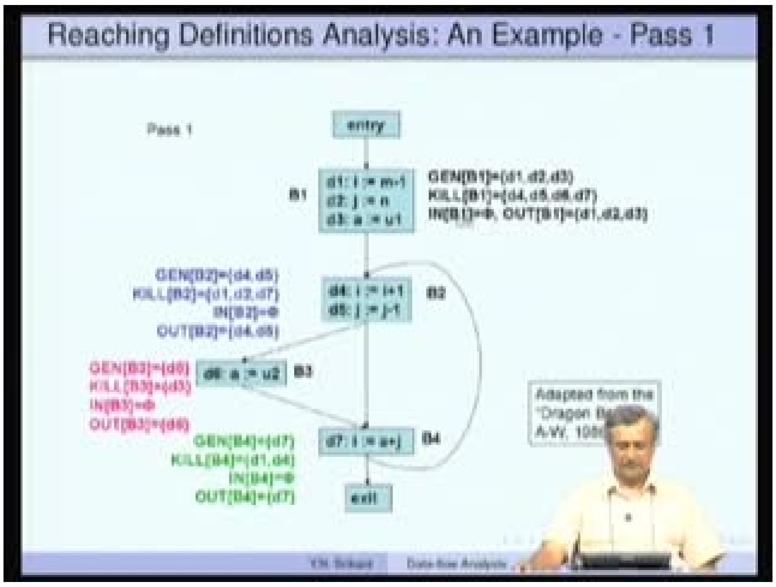
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the ba the flow graph, that are killed by individual s

Y.N. Srikant Data-flow Analysis

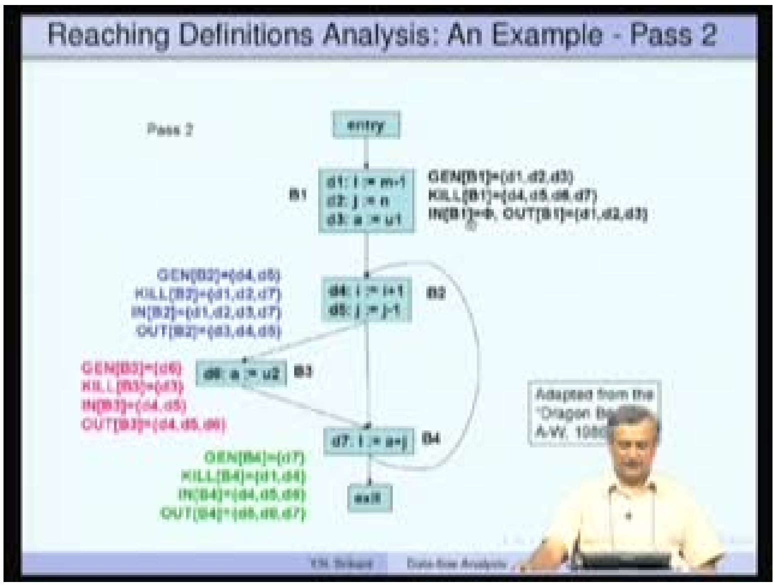
So, we saw in the last lecture, the reaching definitions problem and quickly to recapitulate the reaching definitions - if a definition reaches a particular point then, it implies that from the point of definition to the point at which we are considering there are no more redefinitions for the variable involved in the definition. These are the data flow equations, there are two quantities GEN[B] and KILL[B] here. GEN[B] is the set of all definitions, this is the local reaching definitions, and set of all definitions inside B that are visible immediately after the block and these are downward exposed definitions. KILL[B] is the union of the definitions in all the other basic blocks of the flow graph that are killed by the individual state machine B.

(Refer Slide Time: 02:30)

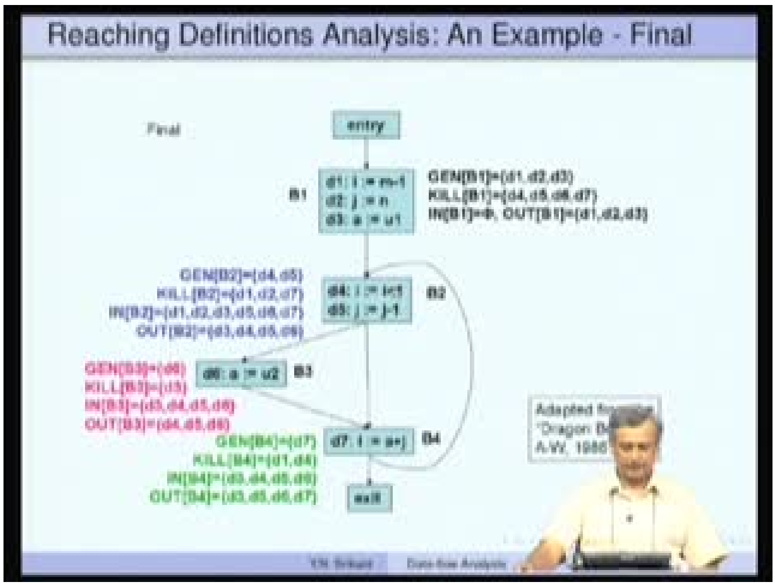


The inset is a union of the outsets of the predecessors of the basic block. This union is the confluence operator in this data flow problem. The outset is a union of the GEN set and then, whatever is coming from the top IN[B] and remove KILL[B] from it, union with GEN. So, that is what gives you OUT[B], IN[B] is phi for all B as an initialization.

(Refer Slide Time: 02:55)

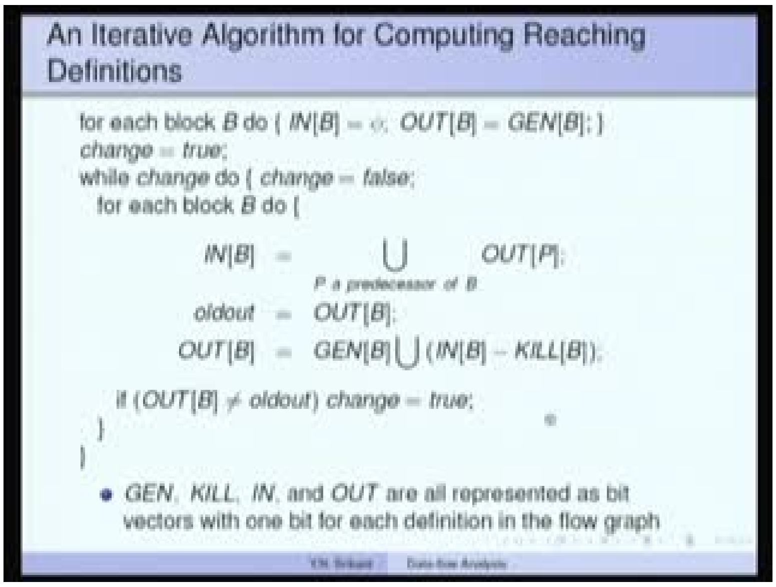


(Refer Slide Time: 03:02)



So, here is the example that we considered in the last lecture. There are 7 definitions here and 4 basic blocks. Here are the GEN and KILL sets and initialization of IN[B1] is phi for all of them. So, OUT [B1] trivially will be GEN[B1] itself, because IN[B1] is phi. Then after one pass, the values become as shown here (Refer Slide Time: 03:00) and the final set of values is in this fashion. If you take for example, B4 what is generated is d7 it’s always so and what is killed by B4 is d1 and d4 because, these are the two definitions of I which appear outside in the basic blocks. IN[B4] is {d3 d4 d5 d6}, so d3 d4 d5 and d6 are the only definitions which reach this point; d1 and d2 do not because when they try passing through B2 they are killed by these two definitions.

(Refer Slide Time: 03:44)



An Iterative Algorithm for Computing Reaching Definitions

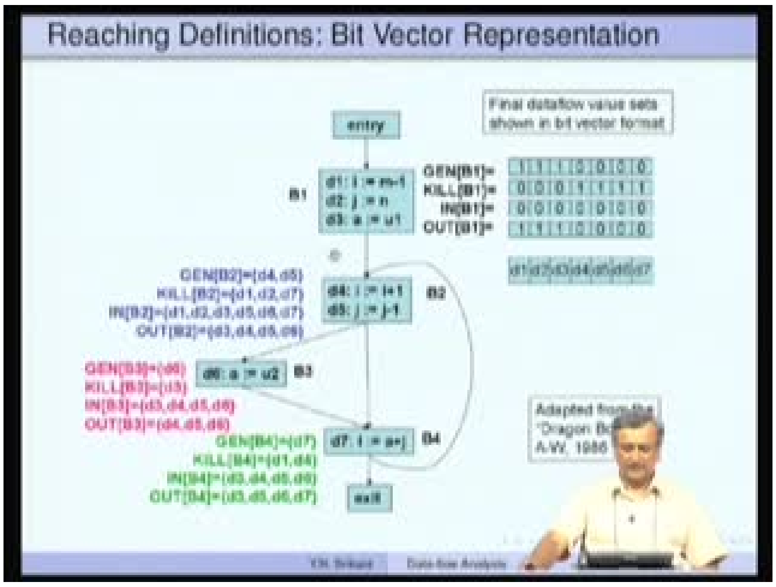
```
for each block B do { IN[B] = $\phi$; OUT[B] = GEN[B]; }
change = true;
while change do { change = false;
  for each block B do {
        $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];$
        $oldout = OUT[B];$
        $OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);$
    if (OUT[B] $\neq$ oldout) change = true;
  }
}
```

- *GEN*, *KILL*, *IN*, and *OUT* are all represented as bit vectors with one bit for each definition in the flow graph

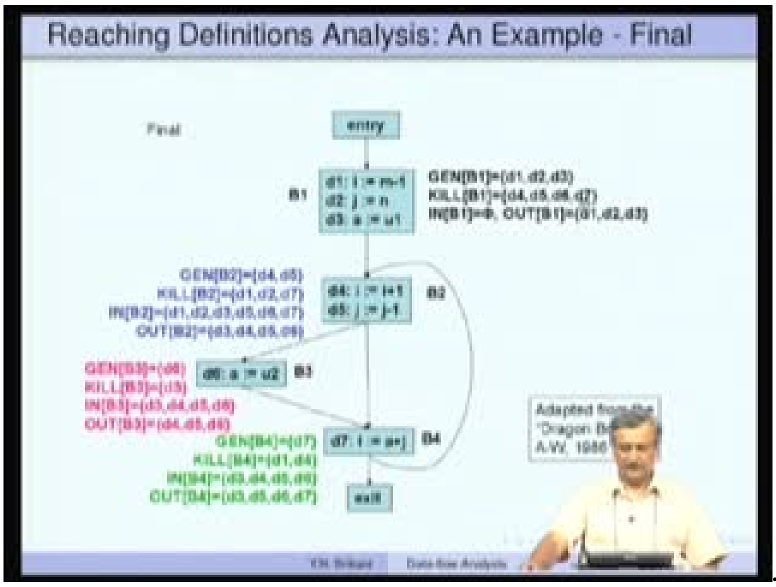
Y.N. Srikant Data-flow Analysis

Now, how do we implement such a data flow analysis? Let us say for reaching definitions, this is actually a fairly straight forward implementation. So, there is while loop, which keeps looping until none of the sets outsets in the system change. So how do we do that? The initialization is of course IN[B] equal to phi and OUT[B] is GEN[B] it could have been phi as well but, GEN[B] reduces the number iterations. There is a Boolean variable change, so while change do; so as to begin with set the changes as false, assuming that we need to iterate once more. Compute IN[B] as given in the equations, so you know union of the out sets of the predecessors, then oldout is retained as OUT[B] and OUT[B] is recomputed as GEN[B] union IN[B] minus KILL[B]. If out B is not the same as oldout then, change the set as true so, we keep going. If change is true then we go back and do it once more, if change is false we then get out. So GEN, KILL, IN and OUT these are all sets, they are represented as bit vectors with one bit for each definition in the flow graph and here is an example to show how it can be done.

(Refer Slide Time: 05:20)



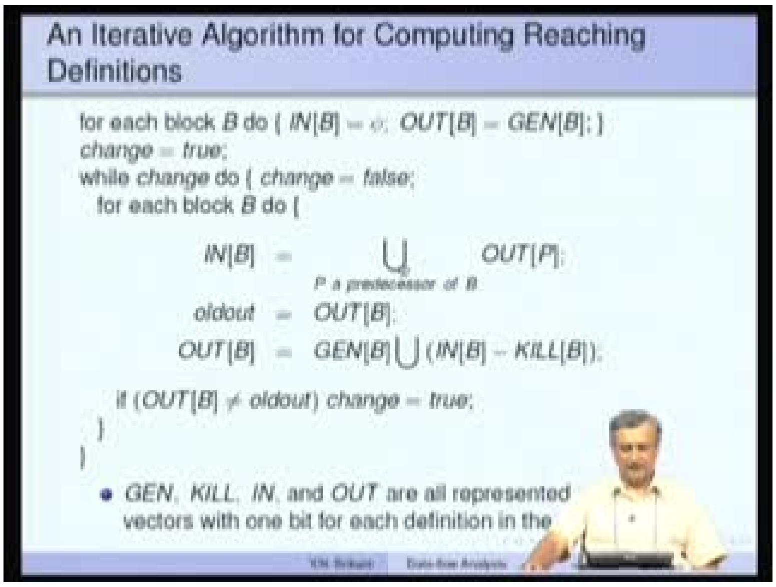
(Refer Slide Time: 05:54)



I have shown this only for the basic block given, but it is going to be similar. There are 7 definitions in this program, so there is one bit for each definition. We have 7 bits necessary to represent this data flow information. Each one of these GEN, KILL, IN and OUT are sets; so, we are representing it as a bit vector. For example, if you look at the previous thing, here GEN[B1] is {d1 d2 d3} and KILL B1 is {d4 d5 d6 d7}. So, d1 d2 d3 implies we set 1s in the three positions corresponding to d1 d2 d3 and then the rest are all

0s. So, d4 d5 d6 d7 are the other definitions, where we set 1s in all these and 0s in these; this is a very simple representation for data flow sets or values.

(Refer Slide Time: 06:36)



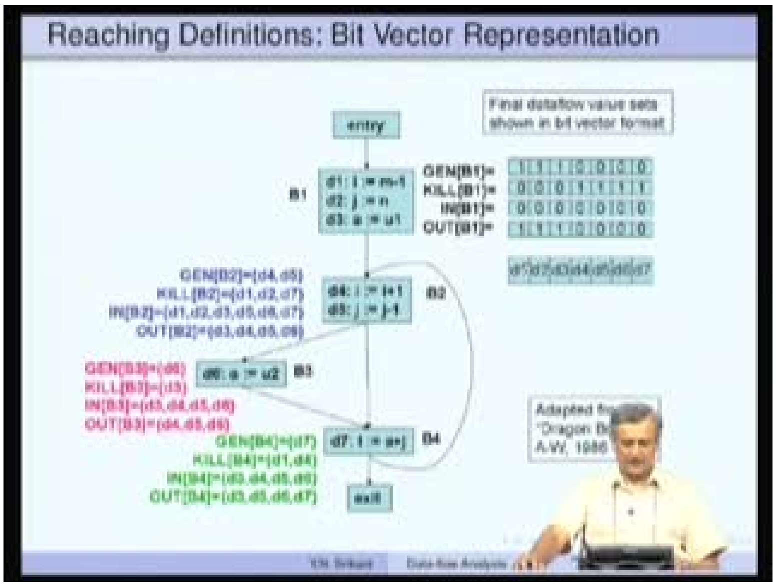
An Iterative Algorithm for Computing Reaching Definitions

```
for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
change = true;
while change do { change = false;
  for each block B do {
        $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$;
     if ($OUT[B] \neq oldout$) change = true;
  }
}
```

- GEN, KILL, IN, and OUT are all represented vectors with one bit for each definition in the

Y.N. Srikant   Data-flow Analysis

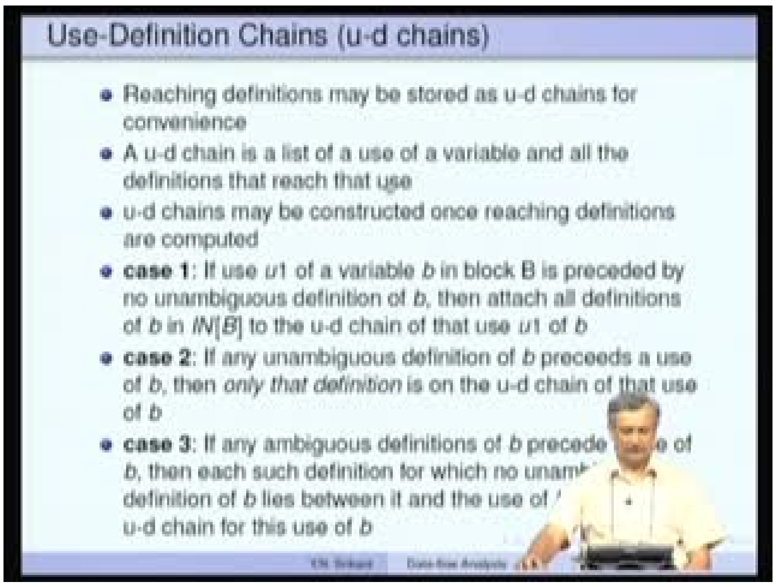
(Refer Slide Time: 06:50)



The advantage of this representation is that, when we want to do a union or intersection or difference operation. For example, here we want to do a union operation here and we want to do a difference operation here, in later cases we may want to do an intersection operation as well (Refer Slide Time: 06:40) . This union operation simply becomes a bit vector OR operation and the difference operation can be similarly implemented in terms of the AND and OR operations. Intersection simply becomes an AND operation. So, the analysis or the program to implement the analysis, reaching definitions analysis, is the very fast because of this bit vector operations provided there is a bit of caution necessary here. The entire set, that is the bit vector of data flow values actually can be fitted, into should be fittable, into a single word of the machine. It could be, if it is a 32 bit machine, then we can have a maximum of 32 definitions and if it is a 64 bit machine then we can have a maximum of 64 definitions fitting into a single word as bits.

In that case a simple AND OR operation you know is possible, but suppose we have 150 definitions in a program, then you know no machine really has 150 or more bits in one single word. So, you may have to use multiple words to implement the bit vectors and then we either have to keep track of the positions of these bit vector part, you know parts as well. So part 1, part 2, part 3, part 4 etcetera and do the AND OR operations appropriately in on all the parts rather than just one word as we see here (Refer Slide Time: 08:39).

(Refer Slide Time: 08:51)



Use-Definition Chains (u-d chains)

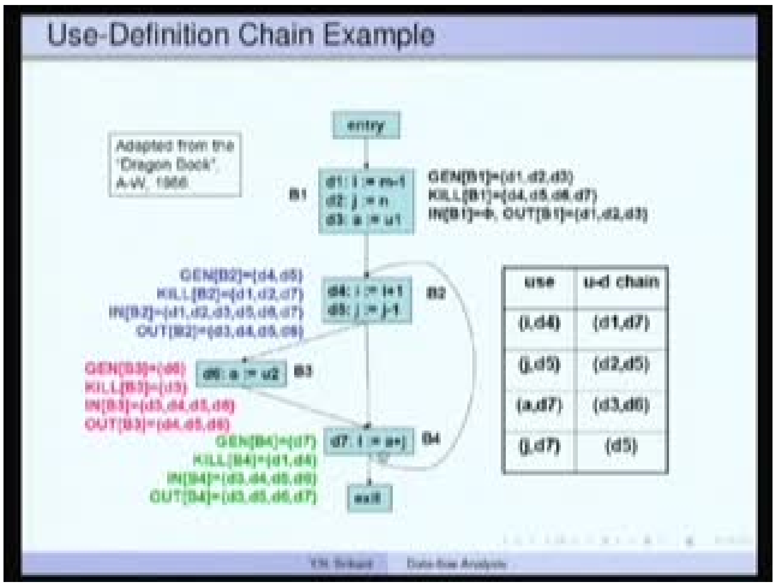
- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1**: If use *u1* of a variable *b* in block B is preceded by no unambiguous definition of *b*, then attach all definitions of *b* in *IN*[*B*] to the u-d chain of that use *u1* of *b*
- **case 2**: If any unambiguous definition of *b* preceeds a use of *b*, then *only that definition* is on the u-d chain of that use of *b*
- **case 3**: If any ambiguous definitions of *b* precede [illegible] e of *b*, then each such definition for which no unam[illegible] definition of *b* lies between it and the use of [illegible] u-d chain for this use of *b*

Y.N. Srikant Data-flow Analysis

Doing this definitely requires a little more work than just one word; so the efficiency goes down a little bit but, there may be no other option as well. This is about the reaching definitions problem. Now, we look at a variation of the reaching definitions problem to compute what are known as used definition chains - u-d chains as they are called.

In fact u-d information is nothing but, reaching definition information. It is just that it is a very convenient data structure to represent such information. What is a u-d chain? It is actually, a list of, a use of variable and all the definitions that reach particular use. So, for each use in the program we are going to attach the definitions that reach that particular that use; let me show you an example quickly.

(Refer Slide Time: 09:38)

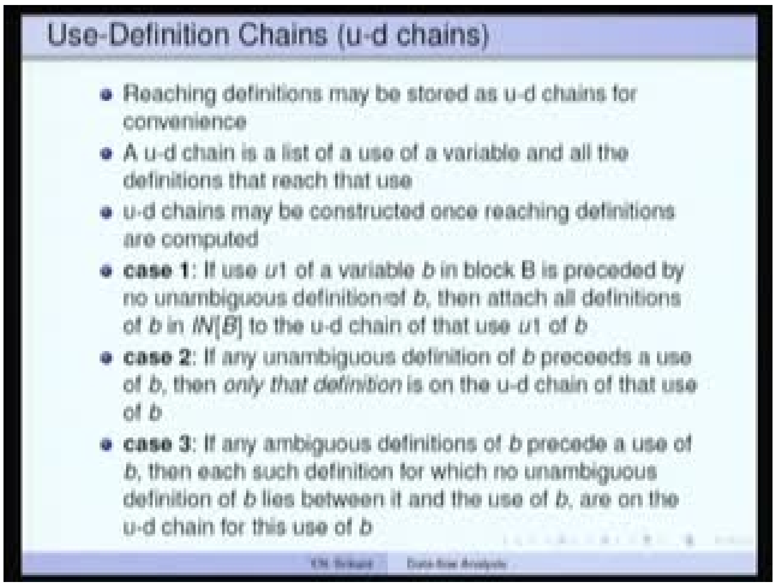

Use-Definition Chain Example

use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	(d5)

So taking the same previous example, there are all the GEN sets, KILL sets, IN sets and OUT sets here. Let us consider i,d4 as I use, the notations says you take the i, which is a use in the definition d4; so, that is our use here. The definitions which reach this particular use, it is very clear d1 reaches it and then d7 also reaches it. The u-d chain for this use will consists of these two definitions d1 and d7 (Refer Slide Time: 10:14). j,d5 is the next use that I have considered; here this is the j and this is the d5, so here d2 of course reaches it and then d5 itself is assigned after the use is completed. This d5 goes all the way takes the back edge and comes all the way to the top of this particular basic block and then reaches this use. Both d2 and d5 use are reachable to this point; whereas, the same is not true for d4 because this d7 kills this d4. So, it is not possible for this d4 to travel all the way up to this point again; that is the reason why we have just d1 and then d7.

a,d7 is here, this is the usage of a,d7 and here we have this d3 which can travel all the way up to this use line, in this path and d6 which can directly travel along this path (Refer Slide Time: 11:23) . So, these are the two definitions which reach this particular use. Finally j,d7 is in the same instruction d7 so only definition which reaches it is d5, that is this particular j (Refer Slide Time: 11:41). This j will not reach this j, this use, because when it tries to pass through this basic block this d5 will KILL d2; so it is not possible for us get that d2 into this place.

(Refer Slide Time: 11:56)

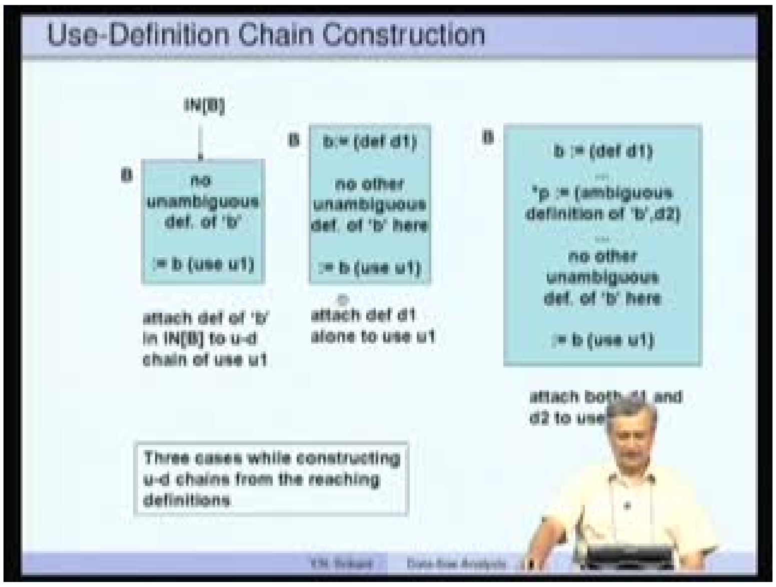


Use-Definition Chains (u-d chains)

- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1**: If use *u*1 of a variable *b* in block B is preceded by no unambiguous definition of *b*, then attach all definitions of *b* in *IN*[*B*] to the u-d chain of that use *u*1 of *b*
- **case 2**: If *any* unambiguous definition of *b* preceeds a use of *b*, then *only that definition* is on the u-d chain of that use of *b*
- **case 3**: If any ambiguous definitions of *b* precede a use of *b*, then each such definition for which no unambiguous definition of *b* lies between it and the use of *b*, are on the u-d chain for this use of *b*

Let us see, how to compute such a data structure. So, u-d chains may be constructed once using the reaching definitions is computed. There are 3 cases, let me show you a picture which corresponds to these 3 cases and then we will read through the information given here.

(Refer Slide Time: 12:16)



Use-Definition Chain Construction

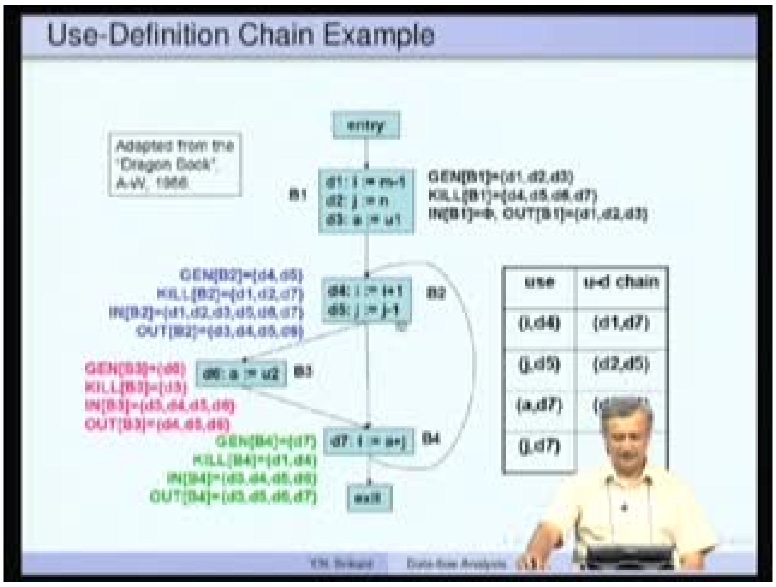
IN[B]
B
no unambiguous def. of 'b'
:= b (use u1)
attach def of 'b' in IN[B] to u-d chain of use u1
B
b:= (def d1)
no other unambiguous def. of 'b' here
:= b (use u1)
attach def d1 alone to use u1
B
b := (def d1)
...
*p := (ambiguous definition of 'b',d2)
...
no other unambiguous def. of 'b' here
:= b (use u1)
attach both d1 and d2 to use

Three cases while constructing u-d chains from the reaching definitions

So, here are the 3 cases: the first case says there is a basic block B, then the inset that is the reaching definitions coming into this basic block have been already computed. There is a use of the variable b, let us say this is called use u1. Then in the rest of this basic block, there are no unambiguous definitions of b. So recall that, unambiguous definition implies, direct definition using a variable; that is something like b equal to and not through a pointer; start p equal to where p may point to the variable b is considered as ambiguous.

We have no unambiguous definitions here, no direct definitions b equal to in this part, so if that is so, the only definitions which can actually come into this basic block are through the set IN. That is why, attach the definition of b in the inset; so there may be many of them here, definitions of B whatever is in this particular IN set are attached to the u-d chain of the use u1. These things they all reach here, because there are no other part definitions to kill the older definitions, so this is one case.

(Refer Slide Time: 13:49)

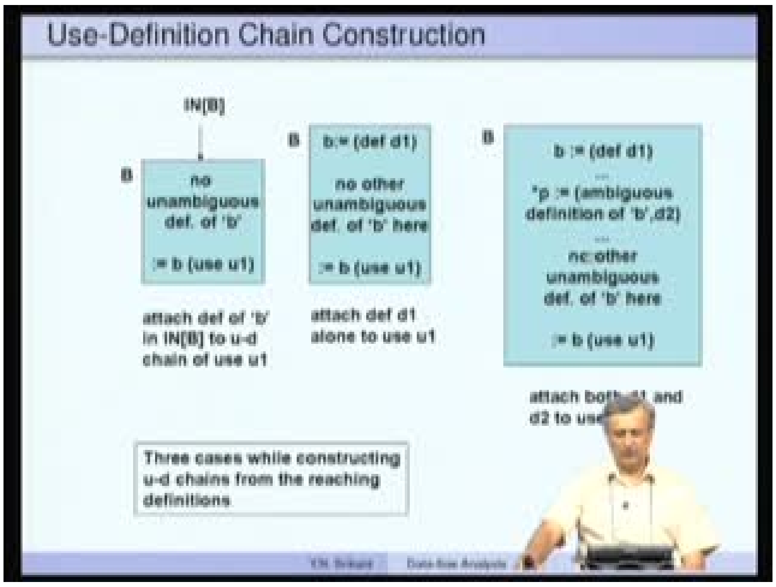


Use-Definition Chain Example

use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	[illegible]
(j,d7)	

We can demonstrate this case very simply here. Let us take i,d4; so, d4 is here and i is here. Now, in this block, there are no other definitions of i before this is used here. So, we look at the inset and in this inset, the definitions of i which correspond to i are d1 and d7 only. So, d1 and d7 are the ones which correspond to i and both are attached to this particular use.

(Refer Slide Time: 14:26)



Use-Definition Chain Construction

IN[B]
B
no unambiguous def. of 'b'
:= b (use u1)
attach def of 'b' in IN[B] to u-d chain of use u1
B
b:= (def d1)
no other unambiguous def. of 'b' here
:= b (use u1)
attach def d1 alone to use u1
B
b := (def d1)
...
*p := (ambiguous definition of 'b',d2)
...
no other unambiguous def. of 'b' here
:= b (use u1)
attach both d1 and d2 to use u1
Three cases while constructing u-d chains from the reaching definitions
YN Srikant
Data-flow Analysis

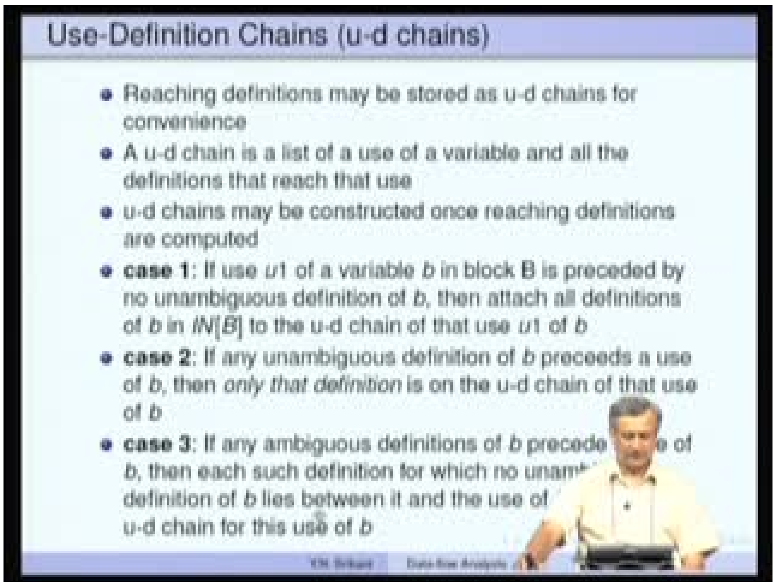
The second case is again there is a basic block and we are looking at the inside of a basic block. So, here is a use u1 of the variable B, then here is a definition d1 of the variable B and in the middle there are no other definitions of the variable B. Again, we are considered only with unambiguous definitions and we are not considering ambiguous definitions, because we do not have to deal with them. Ambiguous definitions do not kill any definition; that is why we do not have to deal with them.

So, for this definition, can straight away reach this use; the value here is what is valid here. Therefore, we can simply attach the definition d1 to the use u1 and put it on its u-d chain. This is the second case; remember that if there were any other definitions which reach through in B like here, they will all be killed by this. So none of them need to be considered, only this definition needs to be considered and that is why attached definition d1 alone to use u1.

The third case is again basic block B; there is a use u1 here, there is a definition d1 here, b equal to and in-between, there is star p equal to which is ambiguous definition of b, that is let us call this as d2. What happens now? Here there were no unambiguous definitions, here also there are no unambiguous definitions but, there is one ambiguous definition. If that is so, you see star p equal to is ambiguous; so p may be pointing to b at the time of execution of this basic block or p may not be pointing to b, at the time of execution of this block.

So we are conservative, we will assume that both the definition, d1 that is this one and the definition d2 are attached to the use u1. This is the conservative nature, there are two of them which are made to actually reach this b. Well, obviously if p is pointing to b at run time then this definition will not reach this point. If p is not pointing to b at run time, this definition d2 will not reach, but d1 will definitely reach. So, either one of them will reach. Since we do not know which one of them is going to reach, we will say both of them reach.

(Refer Slide Time: 17:26)



Use-Definition Chains (u-d chains)

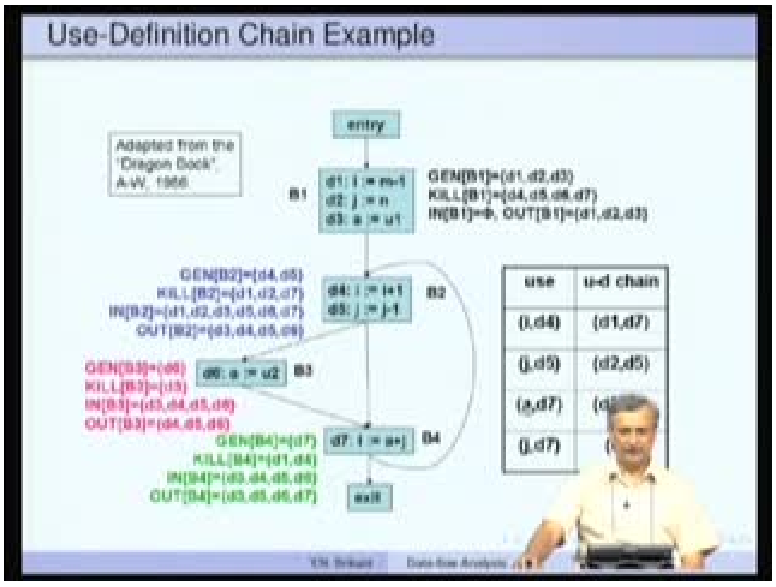
- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1**: If use *u1* of a variable *b* in block B is preceded by no unambiguous definition of *b*, then attach all definitions of *b* in *IN*[*B*] to the u-d chain of that use *u1* of *b*
- **case 2**: If any unambiguous definition of *b* preceeds a use of *b*, then *only that definition* is on the u-d chain of that use of *b*
- **case 3**: If any ambiguous definitions of *b* precede [illegible] of *b*, then each such definition for which no unam[illegible] definition of *b* lies between it and the use of [illegible] u-d chain for this use of *b*

Y.N. Srikant Data-flow Analysis

So, that is what I have said here. Case 1: If use u1 of a variable b in block B is preceded by no unambiguous definition of b, then attach all definitions of b in the set IN[B] to the u-d chain of that use in u1 of b. Case 2: This is very similar, if any unambiguous definition b precedes a use of b, then only that definition is on the u-d chain of that use of b. Case 3: Corresponds to the ambiguous definition that we saw, they both of them are attached to the use of b.

(Refer Slide Time: 18:09)

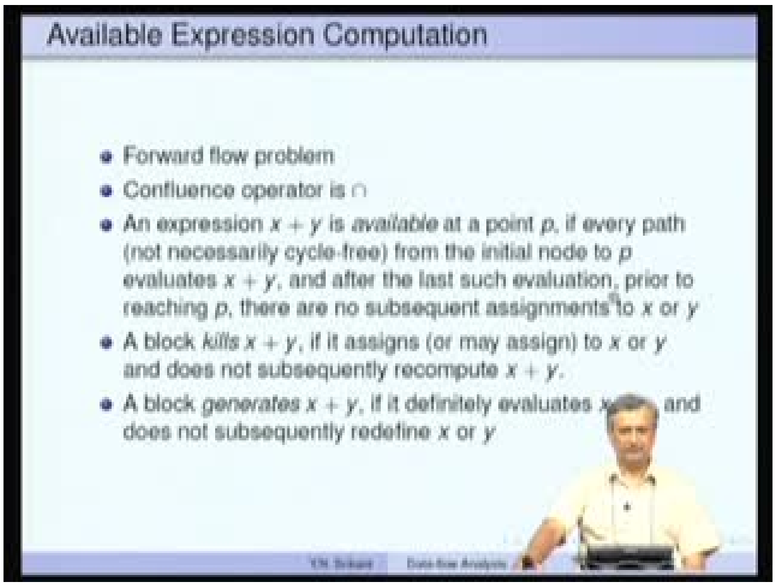
Use-Definition Chain Example



use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	[illegible]
(j,d7)	[illegible]

Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 18:15)



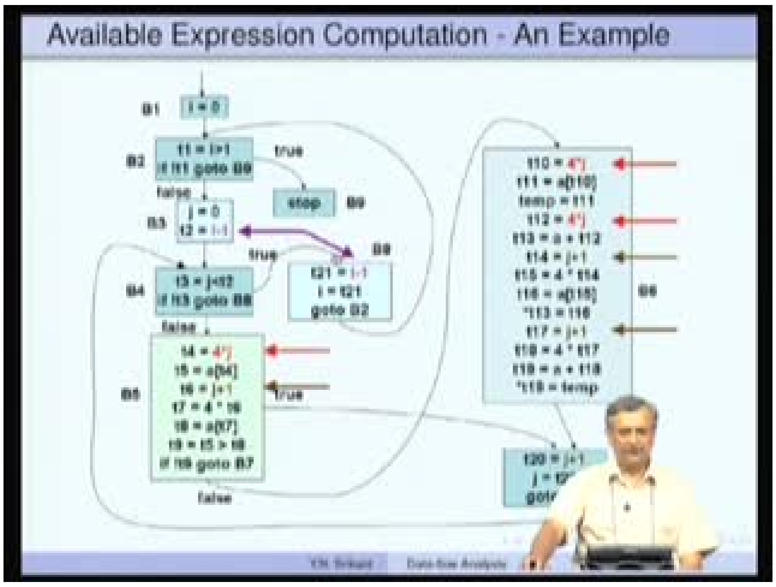
Available Expression Computation

- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$ and does not subsequently redefine x or y

We already saw this particular example. So let us proceed further, let us take up the second data flow computation problem. This is known as the available expression computation and this problem is essential to handle the common sub-expression elimination optimization.

This is also a forward flow problem but, the confluence operator here is intersection. What is meant by availability? Let us define that. It is very simple. An expression x plus y is available at a point p, if every path, not necessarily cycle free, form the initial node to the point p evaluates x plus y, so this is very important, and after such last such evaluation, prior to reaching p, there are no subsequent assignments to x or y. So, this is something I want to show you here.

(Refer Slide Time: 19:35)



Available Expression Computation - An Example

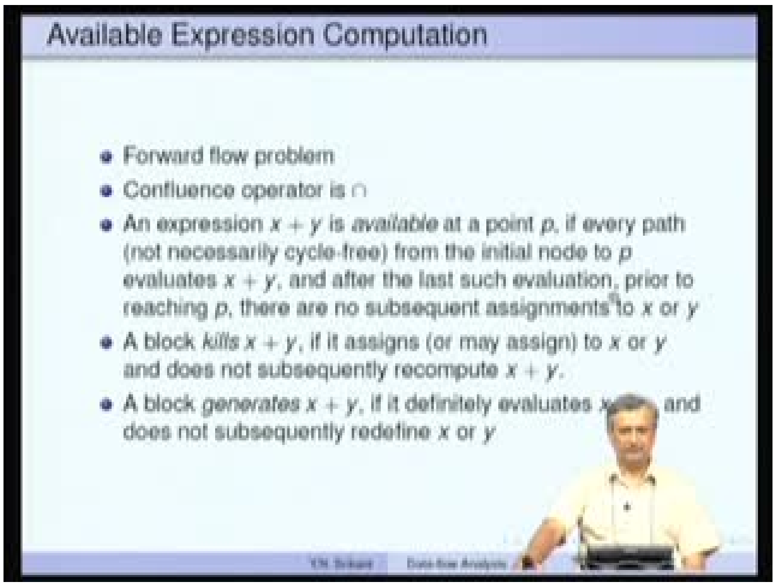
Let us consider the expression 4 star j here; there is another 4 star j here (Refer Slide Time: 19:41). This is the same example that I gave you during introduction to optimizations lecture. There is another 4 star j here (Refer Slide Time: 19:54), we have marked 4 star j by red arrows here; so this 4 star j is said to be available at the entry of the basic block B6. So how is that? When we say 4 star j is available here, we are supposed to look at all the paths from the initial node to this particular point and make sure that there is a computation of 4 star j. Lets trivially make sure of, because every path that reaches here, goes through this basic block. There is no way you can actually reach this point otherwise. Once we go through this basic block, 4 star j is computed. We are not assigning any value to j here, j actually gets assigned only here; so, 4 star j definitely reaches is available at this point (Refer Slide Time: 20:49).

Similarly, j plus 1; so j plus 1 is used here and here (Refer Slide Time: 20:56). If we want to make sure that j plus 1 is available at this point, we must make sure that it is computed along all paths to this point and j is not redefined again, which is obviously true because again as I said, every path that comes here will go through this basic block.

Suppose, you look at i minus 1; i minus 1 is here and i minus 1 is here as well. We want to make sure, that every let us look at availability of i minus 1 at this point. We want to make sure starting from the initial node when before reaching this particular point, the beginning of B8, i minus 1 is evaluated and then i is not redefined. We go through this, so we evaluate i minus 1 and then we can directly reach this; that is one path possible and definitely i minus 1 is evaluated and i is not redefined.

So, we may actually go through this entire thing (Refer Slide Time: 22:02), we go through this particular loop go out, come back again go to the same block B4 and then travel to this point. If we do this we still see that, i minus 1 is evaluated here on this way but, i is not redefined anywhere else.

(Refer Slide Time: 22:51)



Available Expression Computation

- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates x and does not subsequently redefine x or y

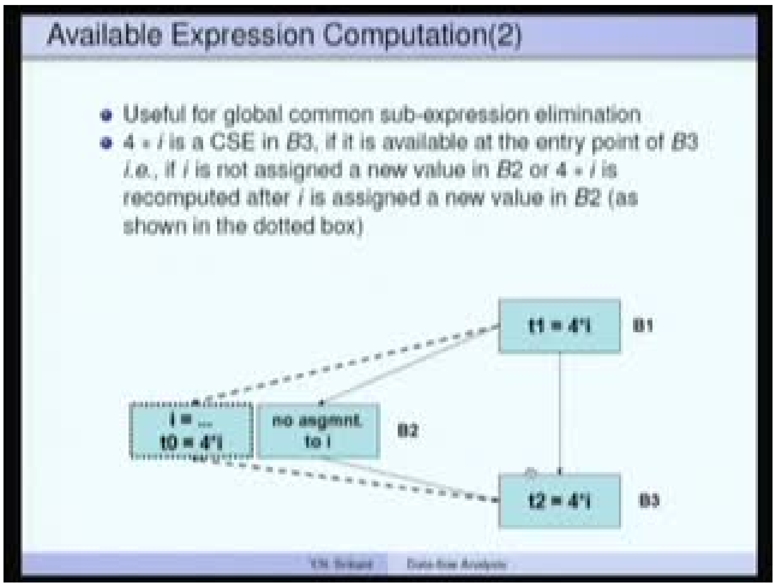
Y.N. Srikant Data-flow Analysis

The third path of course - it comes out like this, goes through this, then takes this true exist goes out and then enters B4 and finally, enters B8. Again, this path also does not have any redefinition of i and i minus 1 is evaluated here; so, i minus 1 is definitely available at this point. This is the meaning of availability, which we demonstrated that this cycle free is not essential, we went through the cycle also.

What about generation and killing of expressions; there is now a pattern in these data flow schemas. We must identify whether it is a forward flow or backward flow problem, we must identify the confluence operator, we must write the constraints and we must write GEN and KILL sets.

What about killing? A block is said to kill an expression x plus y, if it assigns or may assign to x or y and does not subsequently recomputed x plus y. Here, unlike in the case of reaching definitions, the kill part is conservative when it says even may assign is good enough for me to kill. I assign a value to either x or y, either directly as x equal to or y equal to, or through a pointer star p equal to, where p points either x or y. So, even this is taken as an assignment to x or y. If there is an assignment then, you know the previous value of x plus y does not hold again, hold any more. This assignment is said to kill this expression, so that block kills this x plus y.

(Refer Slide Time: 24:36)



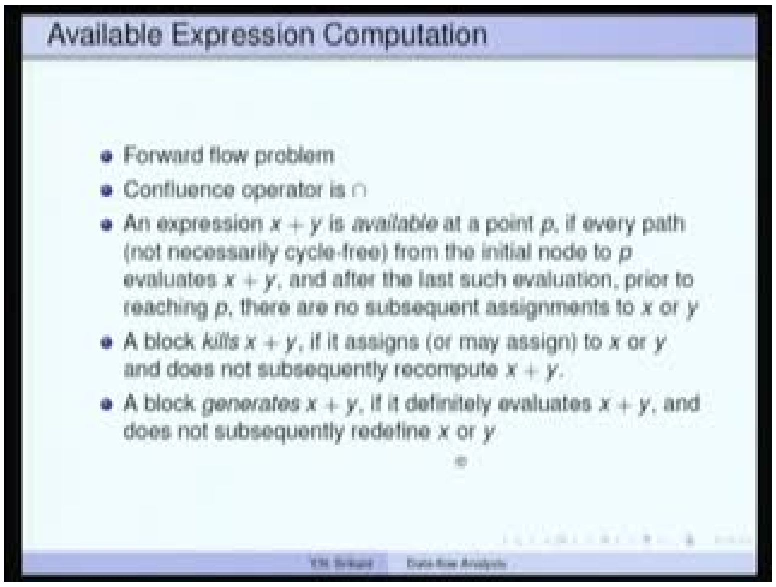
Available Expression Computation(2)

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *i.e.*, if i is not assigned a new value in $B2$ or $4 * i$ is recomputed after i is assigned a new value in $B2$ (as shown in the dotted box)

Of course, there should be no subsequent recomputation of x plus y. Let me make this very clear. Here is t1 equal to 4 star i which is in the block B1, then you know along this path, there is no more redefinition of i. So, 4 star i travels along this path to this point but, what about the other path? There are two possibilities: one is there is no assignment to i; so then we take this solid arrow and go to this block, so if there is no assignment to i, i cannot be modified. So, 4 star i becomes available along this path also.

If we consider the other option there is an assignment to i, but then there is a recomputation of 4 star i. So even in this case, even though i is modified and this value of 4 star i, this expression 4 star i is not valid anymore after this. We recomputed 4 star i and4 star i is said to be available at this point. Please remember that, we are not looking at the value of the expression we are only considering the expression itself. So here is 4 star i, here is 4 star i, this may actually have a different value of i compared to this (Refer Slide Time: 25:42). There is a redefinition of i here, so this 4 star i may be different and this 4 star i may be different but, when we do common sub-expression elimination to eliminate this 4 star i, we simply put this 4 star i into the same variable; say some temp equal to 4 star i and say t2 equal to temp here, here also it will be temp equal to 4 star i. We are not really worried about the actual value of i or the expression but, it is just that the evaluation of 4 star i is carried out along all paths and we do not have to do it more than once if we perform the global common sub-expression elimination.

(Refer Slide Time: 26:30)

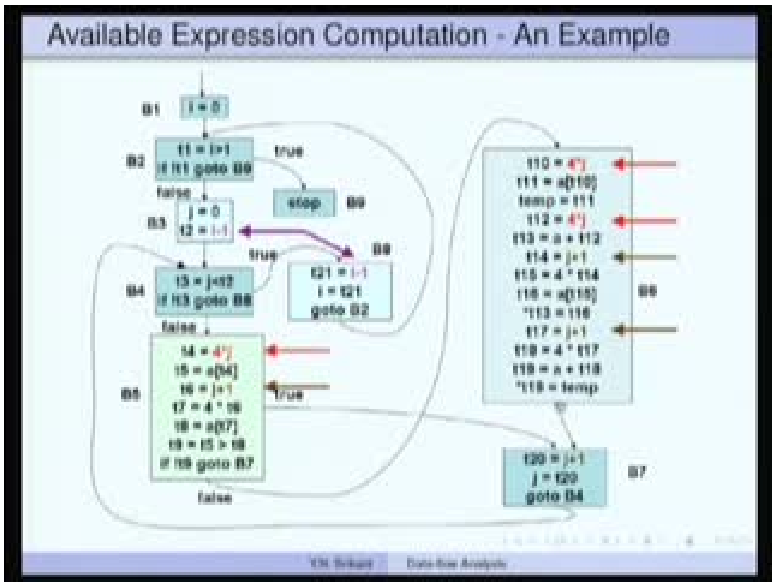


Available Expression Computation

- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 26:48)



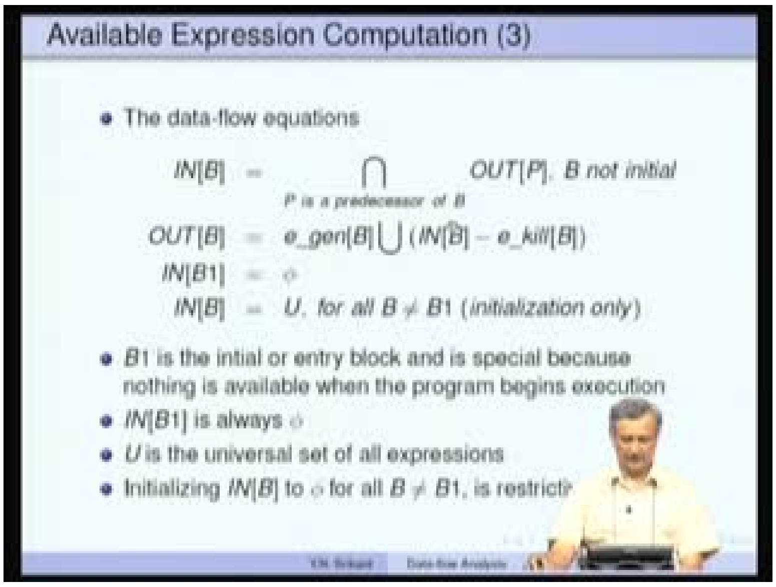
Available Expression Computation - An Example

Y.N. Srikant Data-flow Analysis

What about the generation of x plus y? A block generates x plus y, if it definitely evaluates x plus y and then subsequently does not redefine either x or y, this is very easy to check here. Let us look at the example. So here is a small block, does this block generate i minus 1 (Refer Slide Time: 26:52)? Definitely it does, because it is evaluating i minus 1 and after this there are no more definitions of i. What about this block (Refer Slide Time: 27:00)? The same way, this block does not generate i minus 1, because when i minus 1 is evaluated but, i is redefined here. So, at the end of the block we cannot get this value of i minus 1 i has changed, so i minus 1 might have changed.

What about 4 star j here (Refer Slide Time: 27:25)? 4 star j is evaluated here and in the entire block there are no other definitions of j. So, at the end of this block this GEN set for this block will definitely contain 4 star j. This block will also contain 4 star j because until this point, we have not redefined j. So that is about generation and killing.

(Refer Slide Time: 27:57)



Available Expression Computation (3)

- The data-flow equations

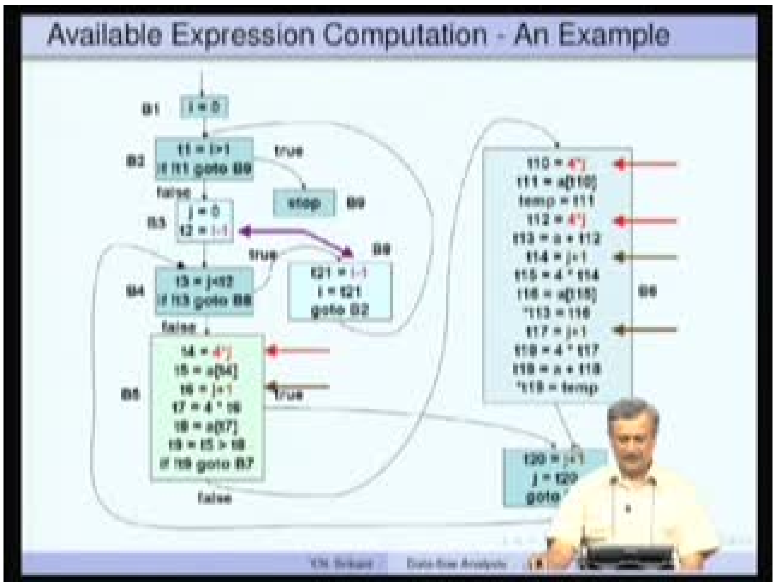
$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P],\ B \text{ not initial}$$
$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$
$$IN[B1] = \phi$$
$$IN[B] = U, \text{ for all } B \neq B1 \text{ (initialization only)}$$

- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restricti

We already saw this example; now, let us look at the data flow equations. The equations again have there are 4 of them, rather 2of them IN[B] and OUT[B] and the other two equations are really initializations. IN[B1] is phi permanently - it is not going to change, why? The B1 is actually an initial or entry block and is special, because nothing is available when the program begins execution. So that is why IN[B1] no expressions are available on entry to the basic block B1, so this is phi and for the other blocks IN[B] is made as U, where U is the universal set of all the expressions in the program. So, we are considering one procedure as a program here and B not equal to B1, for B1 phi is the final value.

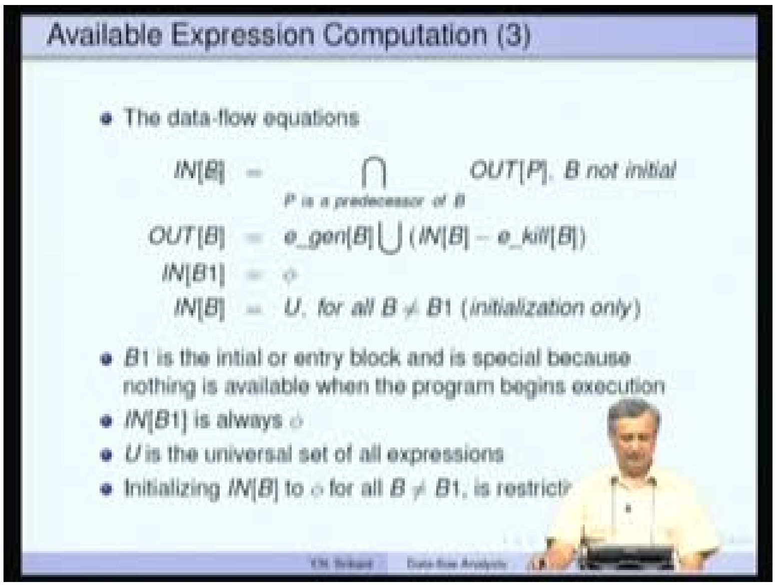
Why should IN[B] be U and why should it not be initialized to phi? We will see that U is always better than phi with the help of an example a little later. Why is the confluence operator intersection? That is because this equation has intersection of OUT[P] where, P is the predecessor of B. So, this is the confluence operator intersection. Why is this forward flow problem? This equation has OUT[B] and IN[B], OUT[B] is being computed in terms of IN[B]. I already mentioned that e_ gen and e_ kill are constants, they do not vary as the program executes, so that is why this is a function of IN[B]. So, a forward flow problem always has OUT[B] as a function of IN[B].

(Refer Slide Time: 30:09)



Available Expression Computation - An Example

(Refer Slide Time: 30:31)



Available Expression Computation (3)

- The data-flow equations

$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P], \; B \text{ not initial}$$
$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$
$$IN[B1] = \phi$$
$$IN[B] = U, \text{ for all } B \neq B1 \text{ (initialization only)}$$

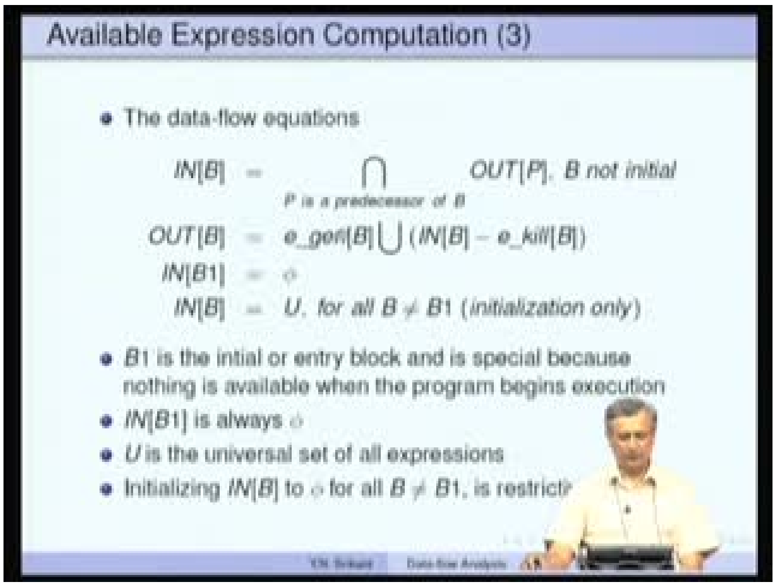
- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restricti

Y.N. Srikant Data-flow Analysis

Let us understand why this is correct? If you are looking at IN[B] it is an intersection of all the OUT[P] sets, P is a predecessor; so let us look at an example here. For example, when we look at a particular basic block, let us take this basic block (Refer Slide Time: 30:15). Here the there is one incoming edge here and another incoming edge here. We want to compute the IN[B] of a basic block. So, let us say we want to compute the IN of this particular basic block B7, then we take the intersection of the outset of this particular basic block and the outset of this particular basic block (Refer Slide Time: 30:43). Why? Then take the intersection of course, why should it be intersection? The intuition is suppose there is an expression which reaches along this path; we start here, it gets evaluated and then it reaches this particular point.

Now, there is couple of expressions which may reach this point like this, along this path (Refer Slide Time: 31:11). Similarly, there are a couple of expressions which go through this path and then reach this point. Which are the expressions? Which definitely reach this particular point? They are nothing but the expressions, which are coming along this path and also this path; that is nothing but, the intersection set.

(Refer Slide Time: 31:59)



Available Expression Computation (3)

- The data-flow equations

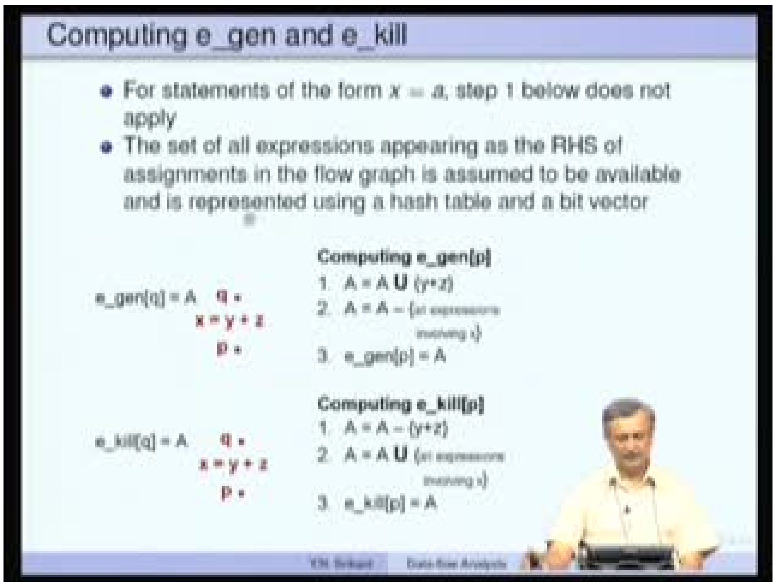
$$
\begin{aligned}
IN[B] &= \bigcap_{P \text{ is a predecessor of } B} OUT[P],\ B \text{ not initial}\\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B])\\
IN[B1] &= \phi\\
IN[B] &= U,\ \text{for all } B \neq B1\ (\text{initialization only})
\end{aligned}
$$

- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restricti

So like in the case of reaching definitions, we do not take union because we want to make certain, that the same expression comes along both the paths. Our requirement of the common sub-expression elimination, tells us that the expression should be reaching along all the paths and not just along one path. So that is the reason why the data flow analysis schema, the equation has intersection. If the application is such that an expression is available, if it reaches along any of the paths coming into the basic block, then this would have also been a union; but, our CSC problem requires that the expression be available along all paths and therefore, it is an intersection.

What about this (Refer Slide Time: 32:27)? Whatever is generated in the block and then union it with, whatever is coming from the top of the block that is the IN[B] and remove what is killed by the block, that is e_kill. So, this is as in the case of reaching definitions, something is generated, something is given to us but, something is being taken away. So, this is as simple as it can be now we also have to look at some details of how to compute e_gen and e_ kill.

(Refer Slide Time: 33:01)



Computing e_gen and e_kill

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A q •
x = y + z
p •

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A q •
x = y + z
p •

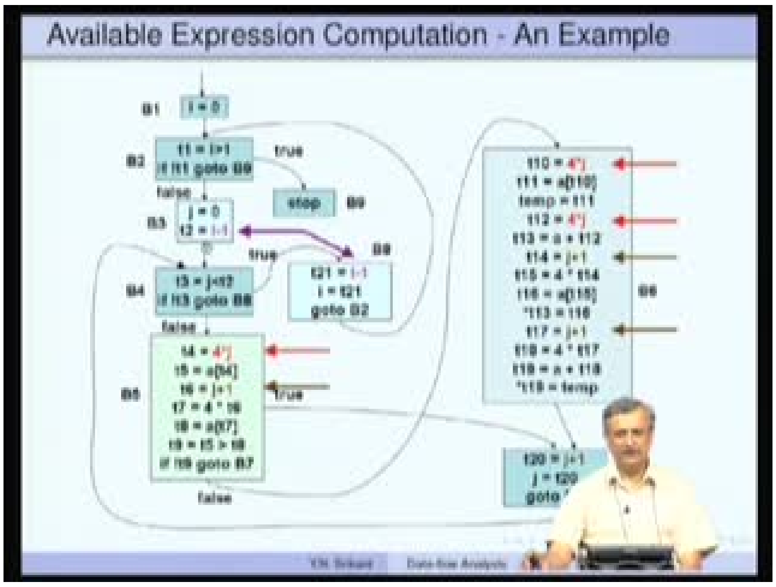
Computing e_kill[p]

1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

Y.N. Srikant Data-flow Analysis

Let us say, the statement is of the form x equal to y plus z here. We are at this point q and when we are given the e_gen at this point q, now we are looking at e_gen computation within the basic block. We have to look at every statement in the basic block see how the e_gen set is computed. To begin with, at this point let us say e_gen is given to us, if it is the beginning of a basic block then this quantity A would be phi nothing.

(Refer Slide Time: 34:49)



Available Expression Computation - An Example

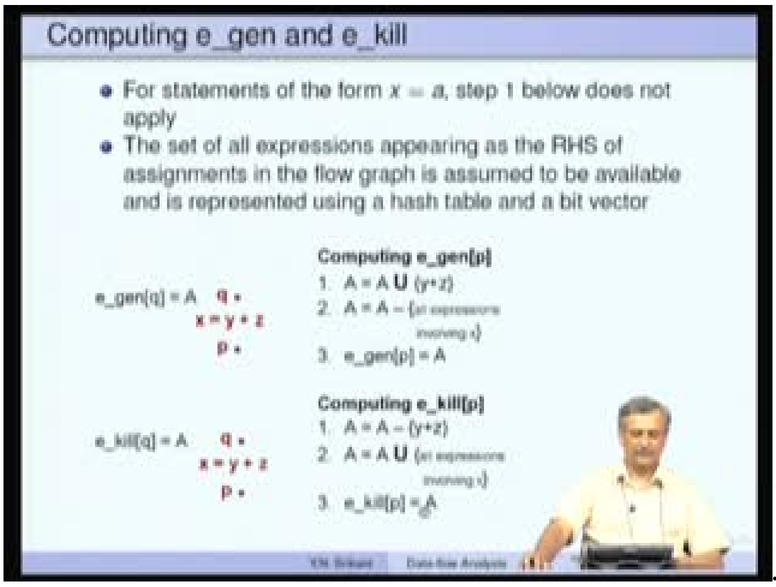
```
B1: i = 0
B2: t1 = i<n
    if t1 goto B9
B9: stop
B3: j = 0
    t2 = i-1
B4: t3 = j<t2
    if t3 goto B8
B8: t21 = i-1
    i = t21
    goto B2
B5: t4 = 4*j
    t5 = a[t4]
    t6 = j+1
    t7 = 4 * t6
    t8 = a[t7]
    t9 = t5 > t8
    if t9 goto B7
B6: t10 = 4*j
    t11 = a[t10]
    temp = t11
    t12 = 4*j
    t13 = a + t12
    t14 = j+1
    t15 = 4 * t14
    t16 = a[t15]
    *t13 = t16
    t17 = j+1
    t18 = 4 * t17
    t19 = a + t18
    *t19 = temp
B7: t20 = j+1
    j = t20
    goto
```

Y.N. Srikant Data-flow Analysis

Now, there is an assignment x equal to y plus z, so that means we are evaluating y plus z. So, this set A now becomes A union y plus z because y plus z is being computed here; but then, we are also assigning to x so all the expressions which had x you know like x plus k x plus y etcetera are all killed by this particular definition of y plus z to x. So, from A you remove all the expressions involving x, so that gives us the e_gen set at this point. This is what the modification is for this particular statement, this is how we compute e_gen.

What we really do is we take the statements one at a time, start with e_gen equal to phi just before the first statement in the basic block. For example, here if you are computing e_gen for this basic block, e_gen will be phi at this point and then forget j equal to 0 for the present, because that is only a constant here, t2 equal to i minus 1. So, i minus 1 is computed, it is put into that e_gen set and then t2 is a temporary. All the expressions involving t2 are removed from the e_gen set, in this case none (Refer Slide Time: 35:15). So, e_gen at this point after this statement will be just i minus 1.

(Refer Slide Time: 35:23)



Computing e_gen and e_kill

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A   q •
x = y + z
p •

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A − {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A   q •
x = y + z
p •

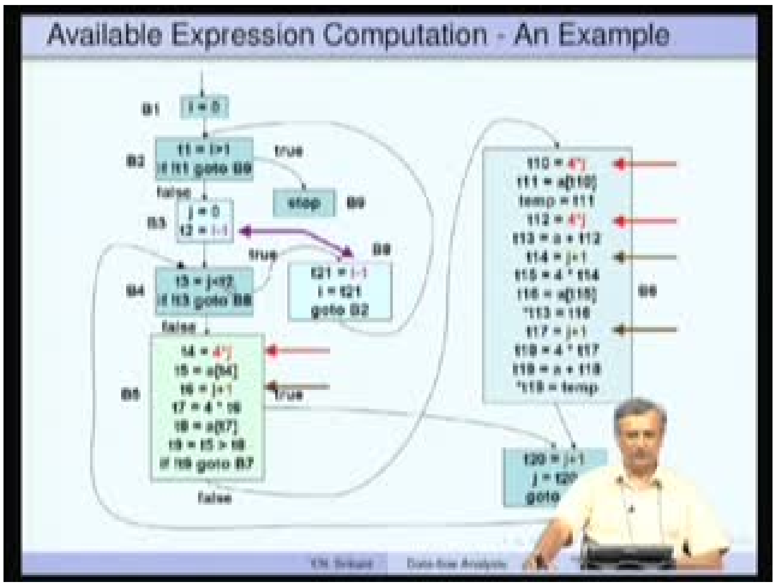
Computing e_kill[p]

1. A = A − {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

YN Srikant   Data-flow Analysis

How about the kill set? It is exactly the other way, whenever you actually compute an expression y plus z; it becomes available, so we are going to remove it from e_kill. It is not killed anymore it is now available, so A equal to A minus y plus z. You have x equal to y plus z, so x is going to kill a lot of expressions involving x; so they are all added to the kill set. A equal to A union all expressions involving x and that is our kill set.

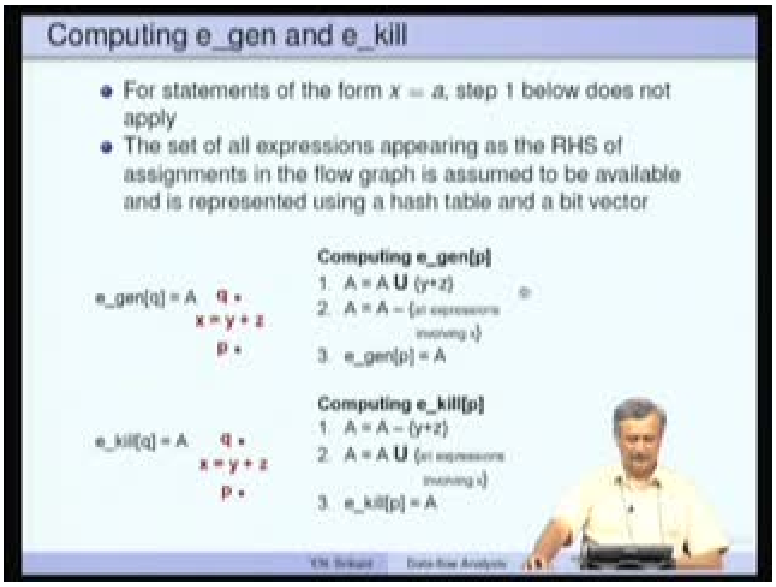
(Refer Slide Time: 36:06)



So, to demonstrate this on this example (Refer Slide Time: 36:06), we have j equal to 0 here. To begin with, suppose the kill set is U for all the expressions; so to begin it is phi. Now involve all the possible expressions in the program which have j. For example, this 4 star j, j plus 1 you know these are the two expressions which we have, so both these are put into the kill set here.

Then we have t2 equal to i minus 1, i minus 1 is now available; so it is removed from the kill set of this basic block and t2 equal to obviously kills all the expressions involving t2. They are all added to this particular e_kill. So, the expressions involving t2are actually very few, so it is only used here; j less than t2 and so on, there are no other expressions involving t2.

(Refer Slide Time: 37:15)



Computing e_gen and e_kill

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A q •
x = y + z
p •

Computing e_gen[p]
1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

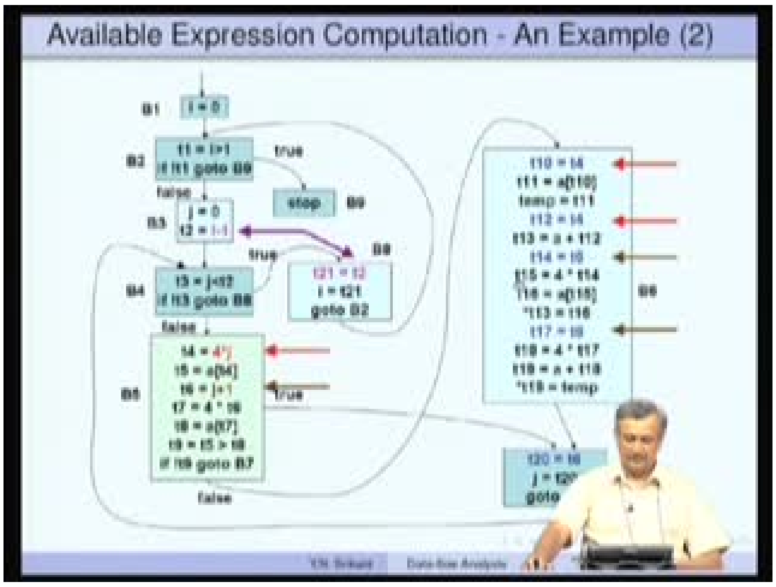
e_kill[q] = A q •
x = y + z
p •

Computing e_kill[p]
1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

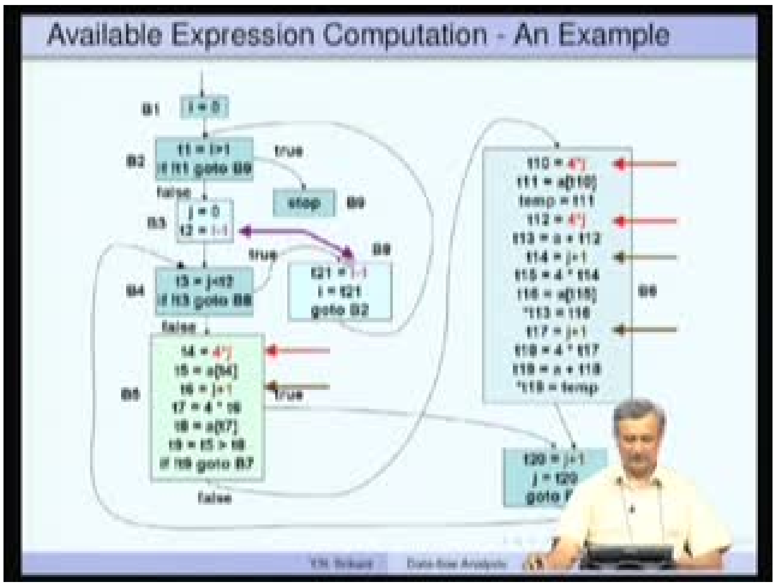
Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 37:24)

Available Expression Computation - An Example (2)



(Refer Slide Time: 37:34)

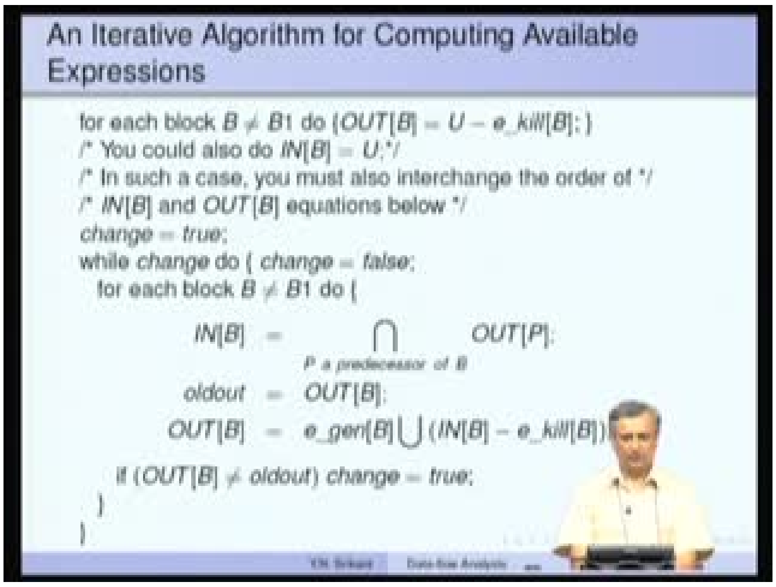


Available Expression Computation - An Example

So, that is the computation of gen and kill and we already saw the equations. Now let us look at, what happened after the available expression computation in this particular example. Since, i minus 1 was available at this point, we could simply take this particular t2 and replace i minus 1 by t2; so that is the common sub-expression elimination that we did here. Similarly, we have 4 star j in t4 here, so it is also the same 4 star j that is here; so we can simply say replace this 4 star j and this 4 star j by t 4, so that we did here (Refer Slide Time: 37:50).

The same is true for j plus 1, so t6equal to j plus 1; so we have replaced the j plus 1 at these points by t6. So, this is the GCSC that we did after the available expression, computation itself.

(Refer Slide Time: 38:24)



An Iterative Algorithm for Computing Available Expressions

```
for each block $B \neq B1$ do {$OUT[B] = U - e\_kill[B]$; }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
change = true;
while change do { change = false;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$;
    if ($OUT[B] \neq oldout$) change = true;
  }
}
```

Y.N. Srikant   Data-flow Analysis

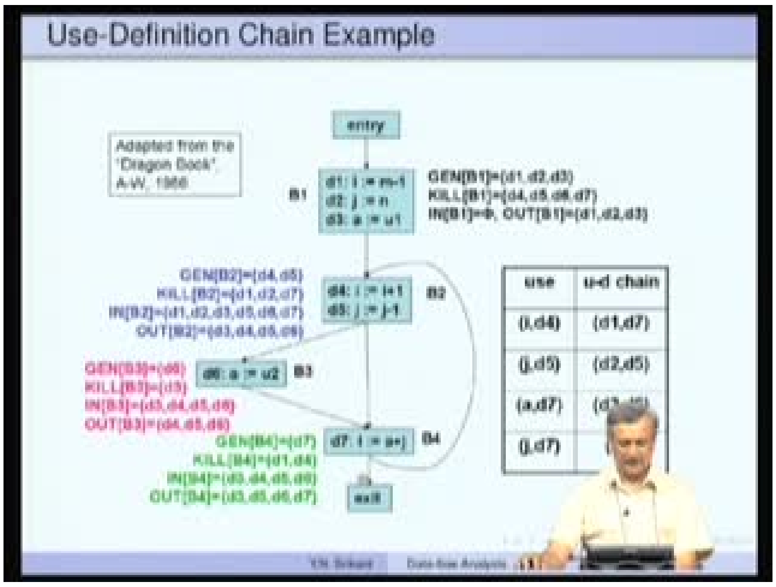
How do you implement this available expression computation? As usual the data flow analysis requires a while loop, it is very similar to the loop that we saw before and the initialization is slightly different. So, for each block B not equal to B1, OUT[B] equal to U minus e_kill. We could also have done IN[B] equal to U, so there are no problem as such and if we had done this, then these two should have been computed in a different order.

The reason is if you know OUT[B] then, you can compute the intersection whereas, if you know IN[B] then, you will have to compute OUT[B] first; that is the reason why the interchange is necessary. Otherwise, the final results will all be the same.

Now, change is set to true and while change do, so first of all make change false saying that, there is no change in the any of the values. So, we may get OUT hopefully; IN[B] is computed form the equation as intersection of OUT[B] we store the oldout value as OUT[B] and OUT[B] is computed fresh as in gen e_gen union IN[B] minus e_kill. Now, if the new value of OUT[B] is not the same as the old value of OUT[B] then, we set change equal to true, it was set false here, so iterate once more.

Once the outsets have all stabilized, insets would have automatically stabilized; so we get OUT and those are our stable values of IN and OUT. So, that is how we do this and again I must mention here, how exactly the implementation of the bit vectors happens? It is very easy to see that, an expression cannot get into a bit. We said d1, d2, d3, d4 etcetera where all definitions; so each one of these was given a bit, if there is a definition in a set, then that was set to one; if the definition is not in a particular set, then the bit was set to 0. Here also we would like to do that for each expression; if there are 20 expressions then, we want to assign one bit to each of these expressions and then say if that expression is in the set, then the bit is set to 1 otherwise set to 0 etcetera.

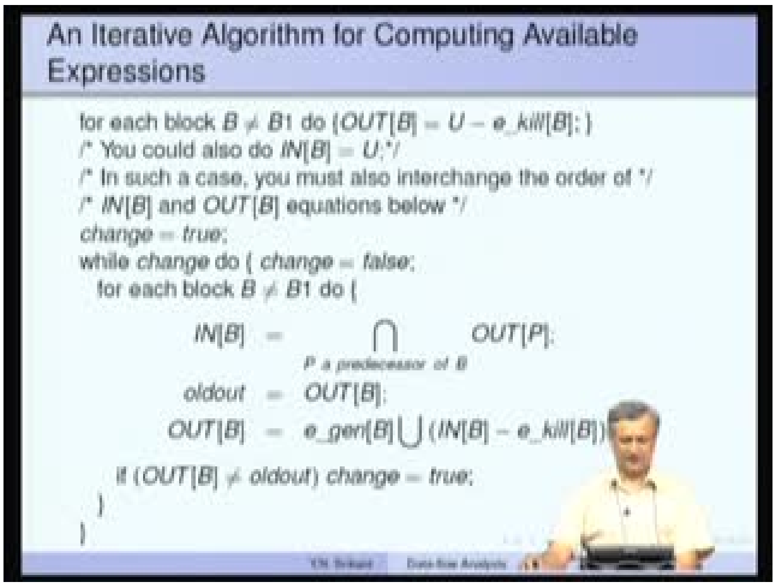
(Refer Slide Time: 41:18)



Use-Definition Chain Example

use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	

(Refer Slide Time: 41:32)



An Iterative Algorithm for Computing Available Expressions

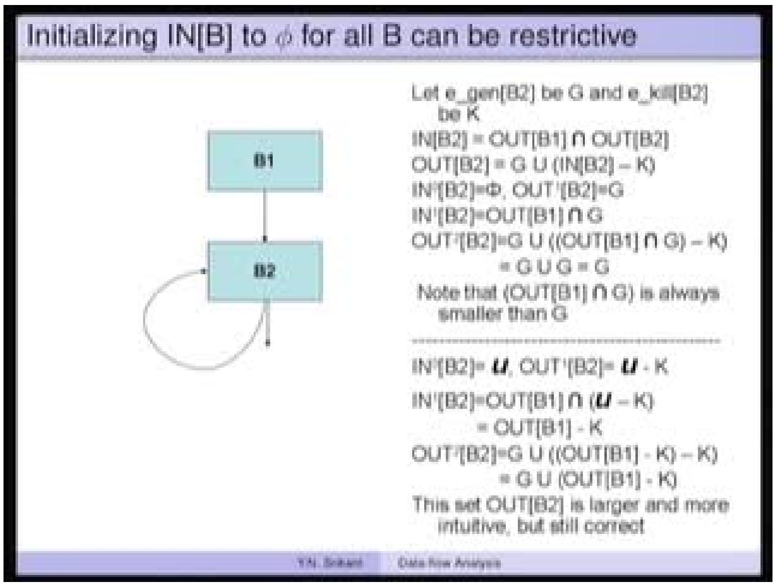
```
for each block $B \neq B1$ do {$OUT[B] = U - e\_kill[B]$; }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
change = true;
while change do { change = false;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$;
     if ($OUT[B] \neq oldout$) change = true;
  }
}
```

But how do we assign unique bit numbers. There, we just numbered took the quadruple number of the definitions. Remember here for example, let us look at this, we have numbered the quadruples d1, d2, d3, d4, d5, d6, d7 so these are the bit positions, we have assumed. Whereas, in the case of assign available expression computation, there is no number given to any of these expressions. It is possible that the same expression is evaluated many times. So, a unique number cannot be provided using the quadruple numbers; so we use the hash table.

What we do is, the first time we meet an expression; we actually put into a hash table, we hash the expression, go to that particular index and use that index value as the bit number. If the same expression is met again and again; we know the same bit number can be used again and again. Suppose the hashing has this problem of collision, more than one expression can hash to the same location. If the hashing function gives you the same location for a different function then, we use one of the collision resolution methods - may be linear collision, quadratic collision, resolution etcetera. Get a new index for that particular expression and use that index as the bit position for that expression.

Somehow using hashing, we must map an expression to a bit position and that make sure that we can use the same bit vector operations. Here we have intersection; here we have set difference and we also have the union, so these can be implemented very efficiently and remember that the bit position is the index of the hash table, so by going back to the hash table, we can also retrieve the actual expression itself.

(Refer Slide Time: 43:22)



Initializing IN[B] to ϕ for all B can be restrictive

Let e_gen[B2] be G and e_kill[B2] be K

$IN[B2] = OUT[B1] \cap OUT[B2]$

$OUT[B2] = G \cup (IN[B2] - K)$

$IN^0[B2] = \Phi$, $OUT^1[B2] = G$

$IN^1[B2] = OUT[B1] \cap G$

$OUT^2[B2] = G \cup ((OUT[B1] \cap G) - K)$

$= G \cup G = G$

Note that $(OUT[B1] \cap G)$ is always smaller than G

$IN^0[B2] = \boldsymbol{U}$, $OUT^1[B2] = \boldsymbol{U} - K$

$IN^1[B2] = OUT[B1] \cap (\boldsymbol{U} - K)$

$= OUT[B1] - K$

$OUT^2[B2] = G \cup ((OUT[B1] - K) - K)$

$= G \cup (OUT[B1] - K)$

This set OUT[B2] is larger and more intuitive, but still correct

Y.N. Srikant   Data flow Analysis

Then, in the available expression one small point was left out. We actually wanted to- We initialized IN B to the universal set remember. So we could do IN B equal to U or OUT B equal to U minus kill, why are we doing this? We must demonstrate that initializing IN B to the universal set is much better than initializing IN B to phi, so to do that let us give an example, take a very small flow graph B1 B2; B2 has a help loop. Let e gen B2 be G, so e gen of B2 is G, e kill of B2 is K. Now, let us see how to compute the IN of B2, you take the intersection of the two sets OUT B1 intersection OUT B2.

So OUT B2 is again form the equation, e gen that is G union IN B2 minus k. So, OUT B 2 is e gen union IN B2 minus whatever is killed by this - that is K, e kill is B2 is K. Now, let us look at the various iterations of this loop IN0 of B2, let us say is phi. We initialized the INs to phi instead of U. Now OUT 1 of B2 in this case, because IN B2 is phi nothing can be taken out of it and OUT B2 becomes G.

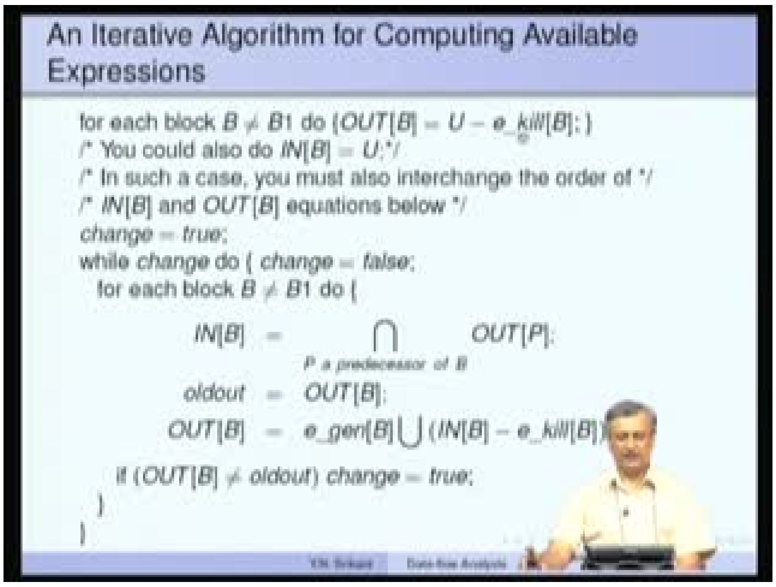
Let us look at the second iteration, IN 1 of B2 is OUT B1 intersection OUT B2, so OUT B2 is G, so OUT B1 intersection G. What about OUT 2 of B2 - second iteration of B2. This is again G union This is IN B2 - that is OUT B1 intersection G is the IN B2 minus K. So, the same equation IN B2 minus K, we have replaced IN B2 by OUT B1 intersection G and here is K. So remember, we are taking an intersection with G, so OUT B1 intersection G can never be bigger than B itself. So this can be safely replaced by G and that is the maximum it can take. So, G union G can be nothing but, just G. This is the maximum that can happen.

Whatever you want, if this had been bigger than G, then G union G prime would have been something bigger but, OUT B1 intersection G, can never be bigger than G. When you take out something from that set G, it cannot be bigger than G either. The maximum that can happen here is G and you get G union G equal to G, so this is OUT B2 and it will not change afterwards.

What about the case, when we initialize IN B2 to the universal set u. OUT B2 will be u minus K, so this is as usual and what about IN B2, we just took OUT B2 as u minus K form the equation. What about IN B2? We have OUT B1 intersection u minus K, so this is OUT B1 - whatever is coming here and this is OUT B2, that is u minus k, the intersection of these two. This is nothing but, OUT B1 minus K because this very simple set theoretic identity. What about OUT2 B2? this is again G union - take the same equations IN B2 minus K; so OUT B1 minus K, which is here minus K, so that is, OUT B1 minus K. You cannot take out the same quantity twice, so this is again set theoretic identity.

Now remember that this is a union here and this particular set is not necessarily a subset of G, it could be anything, because it is out of B1, there may be lot of things inside which are generated and all that. Whereas, G is really the generation of B2 itself, these two are not necessarily related. Whereas, here, in the previous case, there was intersection with G; that is why it was a subset of G - possibly a subset of G.

(Refer Slide Time: 49:12)

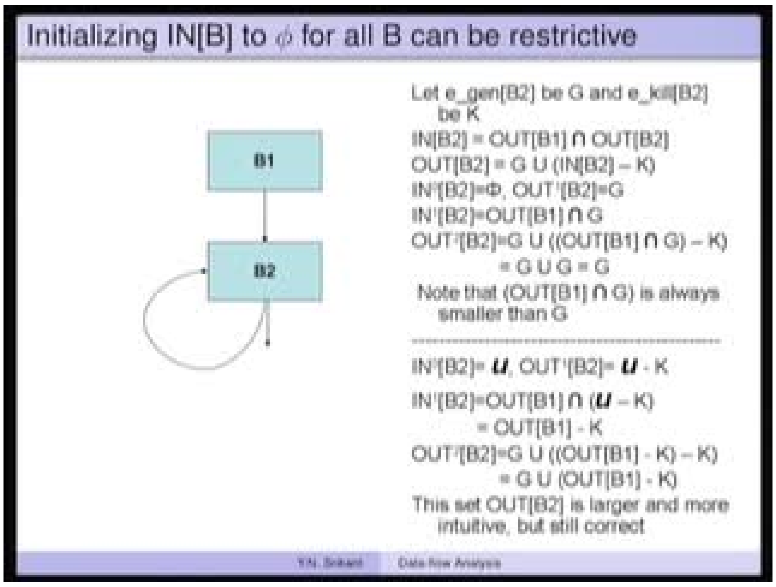


An Iterative Algorithm for Computing Available Expressions

```
for each block $B \neq B1$ do { $OUT[B] = U - e\_kill[B]$; }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do { $change = false$;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$;
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
```

Y.N. Srikant Data-flow Analysis

(Refer Slide Time: 49:19)



Initializing IN[B] to ϕ for all B can be restrictive

Let e_gen[B2] be G and e_kill[B2] be K

$IN[B2] = OUT[B1] \cap OUT[B2]$

$OUT[B2] = G \cup (IN[B2] - K)$

$IN^0[B2]=\Phi$, $OUT^0[B2]=G$

$IN^1[B2]=OUT[B1] \cap G$

$OUT^1[B2]=G \cup ((OUT[B1] \cap G) - K)$
$= G \cup G = G$

Note that $(OUT[B1] \cap G)$ is always smaller than G

$IN^0[B2]= \boldsymbol{U}$, $OUT^0[B2]= \boldsymbol{U} - K$

$IN^1[B2]=OUT[B1] \cap (\boldsymbol{U} - K)$
$= OUT[B1] - K$

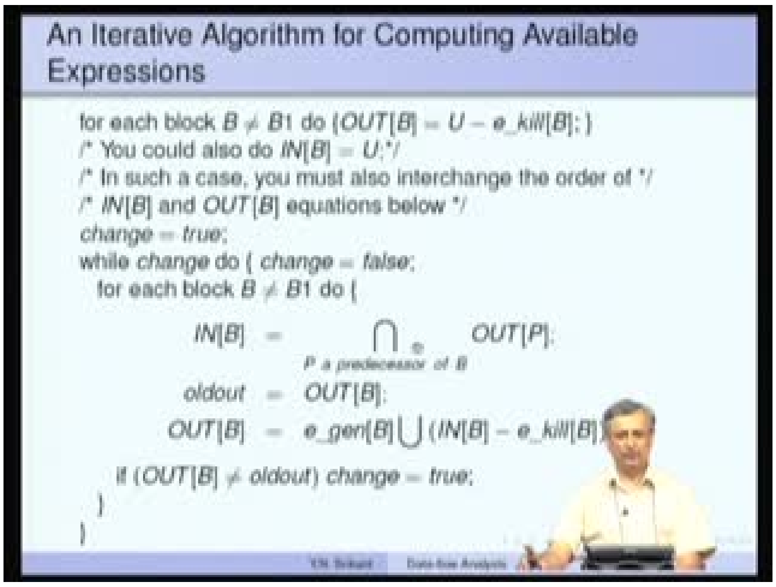
$OUT^1[B2]=G \cup ((OUT[B1] - K) - K)$
$= G \cup (OUT[B1] - K)$

This set OUT[B2] is larger and more intuitive, but still correct

Y.N. Srikant Data-flow Analysis

This set could be different from G, there may be some overlap but, there may be some elements which are not necessarily in G. So because of that this OUT B2 is larger and more intuitive but, still correct. This is larger so it says, whatever is generated within and then whatever is coming here and what is killed - remove what is killed. So this is more intuitive but, at the same time still correct. So the moral of the story is, if you initialize the insets to u or the outsets to u minus k, we actually get a conservative solution which is still better than the other case, when we initialize the insets to phi. So that is the reason why the initialization here was IN B equal to U or OUT B equal to U minus e kill B and not IN B equal to phi as in the case of reaching definitions.

(Refer Slide Time: 49:26)



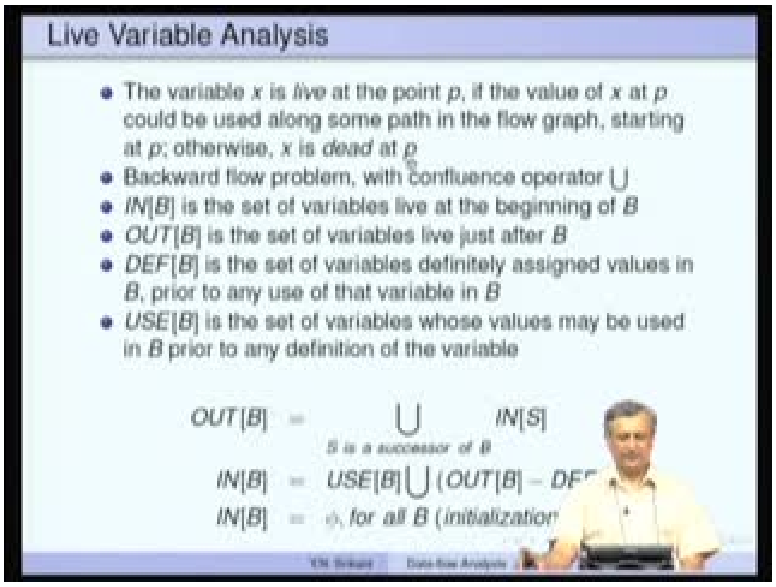
An Iterative Algorithm for Computing Available Expressions

```
for each block $B \neq B1$ do {$OUT[B] = U - e\_kill[B]$; }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
change = true;
while change do { change = false;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$
    if ($OUT[B] \neq oldout$) change = true;
  }
}
```

Y.N. Srikant   Data-flow Analysis

So this is actually a general rule, whenever we have the confluence operator as union; we will have IN B equal to phi and as initialization. Whenever the confluence operator is intersection; we are going to have IN B equal to U as the initialization.

(Refer Slide Time: 49:27)



Live Variable Analysis

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

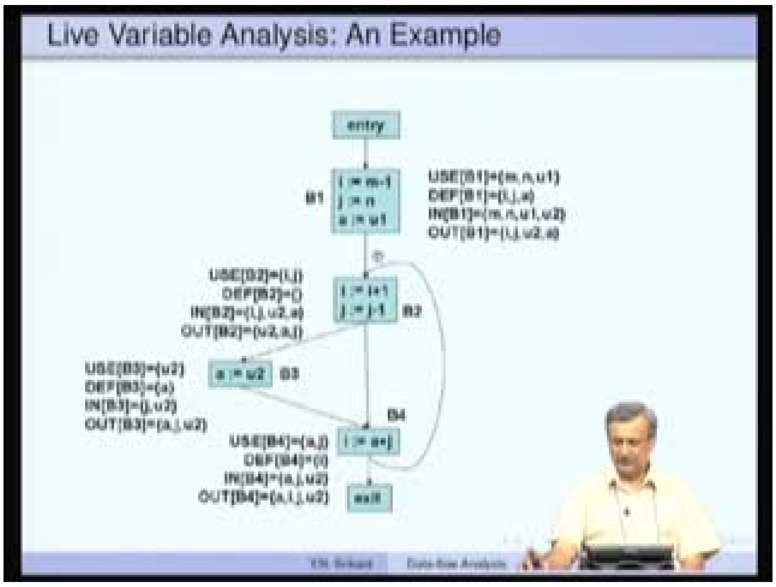
$$IN[B] = USE[B] \bigcup (OUT[B] - DE$$

$$IN[B] = \phi, \textit{ for all } B \textit{ (initializatio}$$

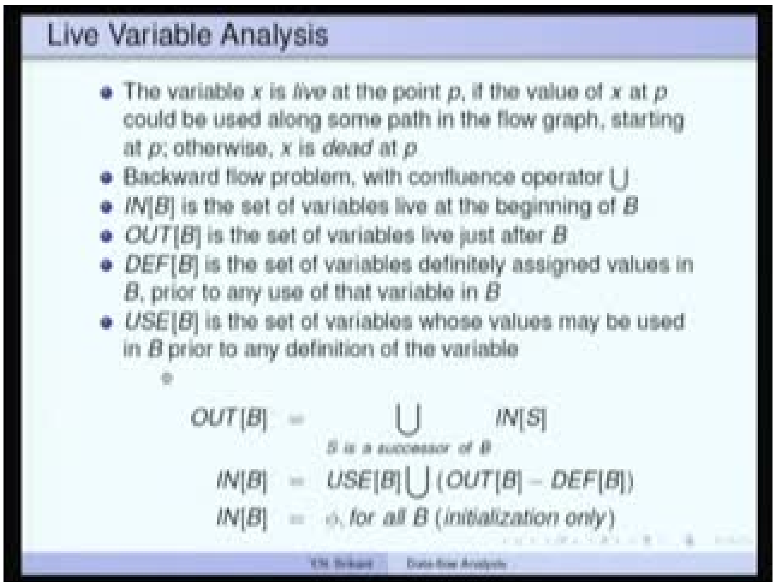
Y.N. Srikant   Data-flow Analysis

So that is about available expression in computation. Now, let us look at the third data flow analysis, which is a very important one and we have already seen the result of this before, Live Variable Analysis.

(Refer Slide Time: 50:44)



(Refer Slide Time: 51:59)



Live Variable Analysis

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$
\begin{aligned}
OUT[B] &= \bigcup_{S \text{ is a successor of } B} IN[S] \\
IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\
IN[B] &= \phi, \text{ for all } B \text{ (initialization only)}
\end{aligned}
$$

Y.N. Srikant Data-flow Analysis

So, we used live variable analysis information during register location. So remember live intervals and live ranges they are all derivatives of this live variable analysis. When do we say a variable is live? So a variable is live at the point p, if the value of x at p could be used along the some part in the flow graph starting at p, otherwise x is dead at p, it is a very simple definition. So for example, suppose we want to answer the question at this point is I live at this point, so I is immediately assigned a value here, so I is not live at this point. Whereas, if you take the question is u1 live, definitely u1 is not assigned any value but, u1 is assigned here along this path.

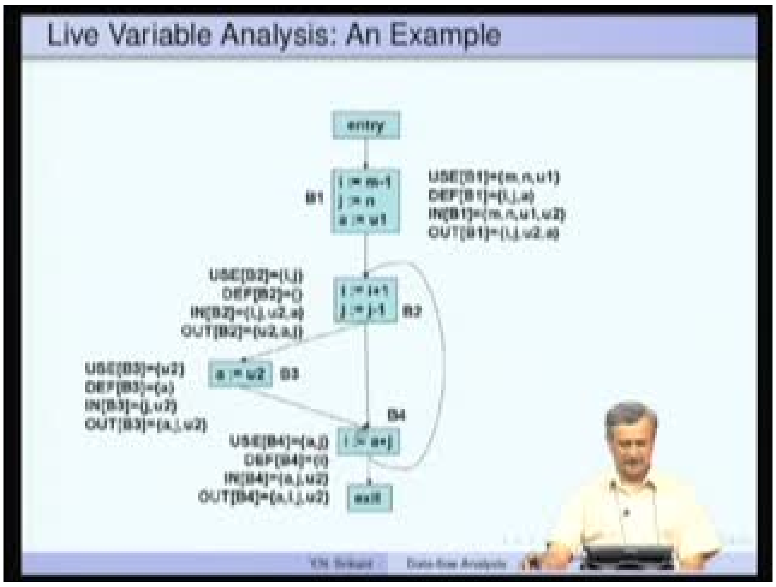
So is u2 live at this point, definitely because I can take this path to u2 and there is a usage of it. So that is how we actually is. if you look- . Let us take this point or even this point; I want to answer the question is u1 live at this point, no matter which path you take this way or you take this way, you will never reach this point, so because of that there is no usage of u1 after we leave this basic block. So, none of the points after this basic block B1 are live, as far as the variable u1 considered. So this is how we compute or define the liveness of a variable.

There must be some usage without redefinition that is the crux of the matter. This is a backward flow analysis problem and the confluence operator is union, so you can see that here. The IN function B set is computed as a function of the OUTB, so IN is a function of OUT that is why, this is a backward flow analysis problem. The confluence operator here is U and cause a consequence of that as I mentioned a few minutes ago IN B is set to phi for all B for initialization purposes. IN B is the set of variables live at the beginning of basic block B, OUT B is the set of variables live just after the basic block B this is as usual.

Instead of gen and kill sets; gen set is USE set and kill set corresponds to DEF set. So what is the DEF or kill of B; DEF set is the set of variables, definitely assigned values in the basic block B prior to any use of that variable in the basic block B.

So, we want to define a variable before using it in the same basic block. USE B is the other way, set of variables whose values may be used in the basic block B prior to any definition of the variable in the basic block B.

(Refer Slide Time: 53:59)

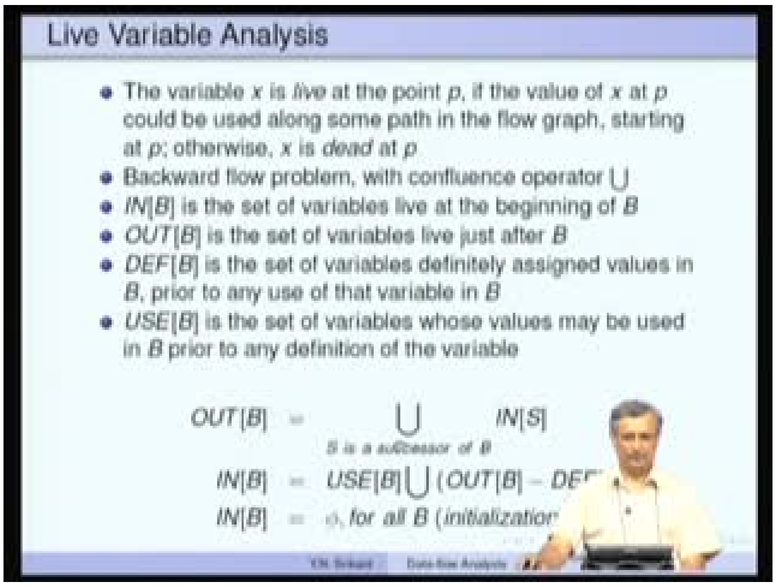


So let us look at the example, to understand what this is. We want to let us say compute the USE and DEF sets for particular basic block B1. We have the variable m, the variable n and the variable u1, which are used in the basic block but, there are no definitions of the three variables before these usages; m is not defined before the usage here, n and u1 are also not defined before their usage here. Therefore, the USE set of the basic block B1 will have m, n and u1. The three variables I, J and a are definitely defined in the basic block, before any use of I, J and a. Therefore, they are put into the DEF set of the basic block B1.

What about basic block B2? This is a little subtle, you must observe that the I here is the old value of I and then, there is a new assignment to I and then onwards the new value of I rings. Similarly, the old value of J is relevant at this point and new value becomes relevant after this J is used, so as far as this I is concerned it is used before a definition of the same I, as far as this J is concerned it is used before a definition of J. So both these are put into the USE set of B2. The DEF set there are only two variables I and J which are assigned values here but, neither of them are defined before any use of that variable see I is used and then it is defined J is used and then it is defined. Therefore, the DEF set of B2 is phi, it is a null set.

What about the basic block B3? The USE set is u2, because u2 is not defined; it is used before its definition and the DEF set is a, because a is defined before it is not used not even used in the basic block. In the case of the basic block B4, we have a and J in the USE set, because they are used before any definition; there is no definition of a and there is no definition of J either but, there is a definition of I and we have not used I, so the DEF of B4 is actually I.

(Refer Slide Time: 56:15)



Live Variable Analysis

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DE$$

$$IN[B] = \phi, \text{ for all } B \text{ (initializatior}$$

Y.N. Srikant Data-flow Analysis

So this is the definition of USE and DEF. The DEF corresponds to the kill set and USE corresponds to the gen set. So, we will stop at this point and take up a detailed discussion of the data flow equations and the example next time. Thank you.