

Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

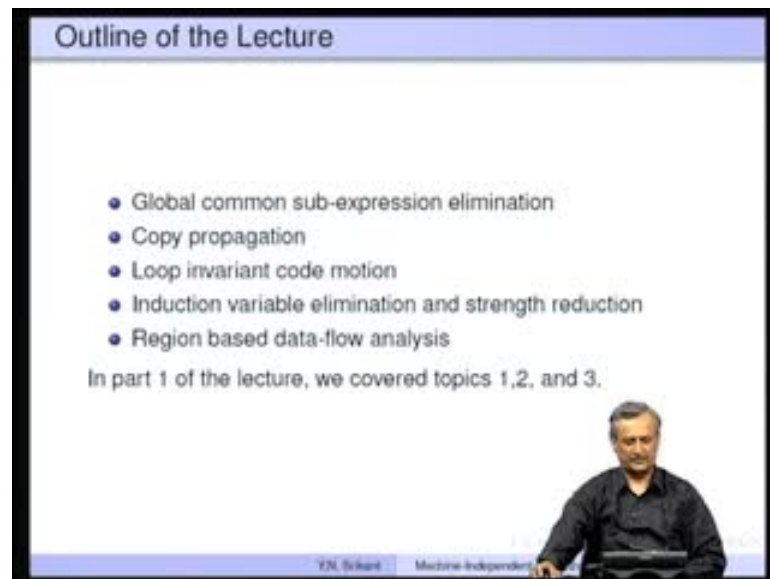
Module No. # 10

Lecture No. # 17

Machine-Independent Optimizations – Part 2

Welcome to the lecture on Machine-Independent Optimizations part 2. In the last lecture, we covered global common sub-expression elimination, copy propagation and loop invariant code motion. Also, I gave you an introduction to what induction variables are. We will continue that lecture today along with region based data-flow analysis.

(Refer Slide Time: 00:21)



Outline of the Lecture

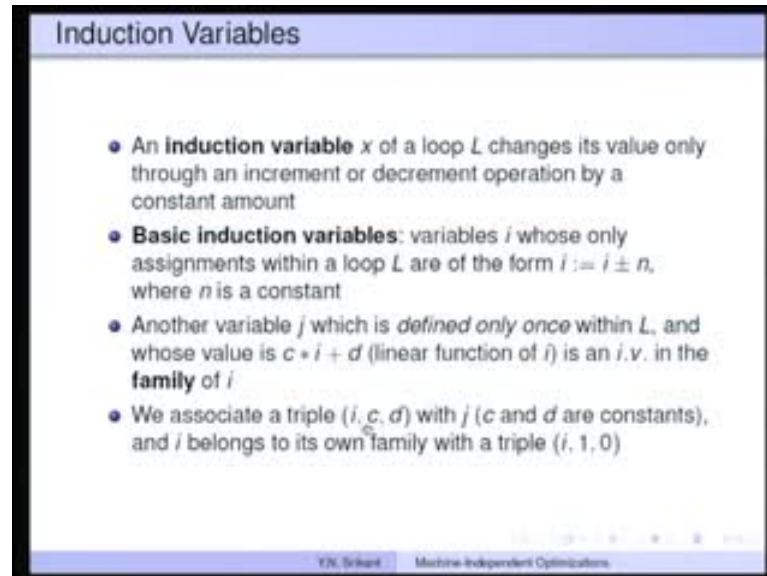
- Global common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Region based data-flow analysis

In part 1 of the lecture, we covered topics 1, 2, and 3.

Y.N. Srikant

Y.N. Srikant Machine-Independent

(Refer Slide Time: 00:42)



The slide, titled "Induction Variables", contains the following text:

- An **induction variable** x of a loop L changes its value only through an increment or decrement operation by a constant amount
- **Basic induction variables:** variables i whose only assignments within a loop L are of the form $i := i \pm n$, where n is a constant
- Another variable j which is *defined only once* within L , and whose value is $c + i + d$ (linear function of i) is an *i.v.* in the **family** of i
- We associate a triple (i, c, d) with j (c and d are constants), and i belongs to its own family with a triple $(i, 1, 0)$

At the bottom of the slide, it says "Y.N. Srikant Machine-Independent Optimizations".

An induction variable is actually related to a loop; essentially, the counter values in a loop or induction variables. The other variables, which are dependent on the loop variables are also induction variables provided they satisfy certain conditions. The basic requirement is – the value only changes through an increment or decrement operation by a constant variable constant amount. There are basic induction variables, which have a form i equal to i plus minus n , where n is a constant. Then, there are derived induction variables such as j , which have the form $c + i + d$ and they are defined only once within the loop; c and d are constants essentially in your functions. If i goes as 1, 2, 3, 4, then j let us say is $5 + i$; then, it goes as 5, 10, 15, etcetera. If it is $5 + i + 3$, then the first value will be $5 + 3$, which is 8 and then $8 + 5$, which is 13 and so on and so forth.

For each induction variable, we have a triple. For a basic induction variable, it is denoted as $(i, 1, 0)$ and any other variable in the family of i is denoted as (i, c, d) ; c and d are the constants. This means, j equal to $i + c + d$; that is how it would be read.

(Refer Slide Time: 02:19)

The slide is titled "Induction Variables - Example 1". It contains the following code:

```
t1 = 202
i = 1
t3 = addr(a)
t4 = t3 - 4
L1: t2 = i > 100
    if t2 goto L2
    t1 = t1 - 2
    t5 = 4 * i
    t6 = t4 + t5
    *t6 = t1
    i = i + 1
    goto L1
L2:
```

To the right of the code, it says: "i is a basic i.v. and t5 is a derived i.v. in the family of i".

A speaker is visible in the bottom right corner of the slide.

Here is an example from the last lecture: i is an induction variable, which is being incremented by 1 in the loop; $t5$ is a derived induction variable in the family of i , which is defined as $4 \times i$.

(Refer Slide Time: 02:38)

The slide is titled "Induction Variables - Example 2". It shows a flowchart with the following blocks:

- B1: $i = m - 1$, $j = n$, $t1 = 4 * n$, $v = a[t1]$
- B2: $i = i + 1$, $t2 = 4 * i$, $t3 = a[t2]$, **if** $t3 < v$ **goto** B2
- B3: $j = j - 1$, $t4 = 4 * j$, $t5 = a[t4]$, **if** $t5 > v$ **goto** B3
- B4: **if** $i > j$ **goto** B6
- B5 and B6 are exit points.

To the right of the flowchart, it says: "i and j are both basic i.v. in both inner and outer loops" and "t2 (in the family of i) and t4 (in the family of j) are both derived i.v. in both inner and outer loops".

A speaker is visible in the bottom right corner of the slide.

Here is another example: i and j are both induction variables, both in the inner loop, this loop, and the outer loop; $t2$ is an induction variable derived from i , which is in the family of i ; again both in this loop and the outer loop. Similarly, j is an induction variable in the

family of t_4 , which is an induction variable in the family of j , both in this loop and the outer loop.

(Refer Slide Time: 03:10)

Detection of Induction Variables

We need a loop L , reaching definitions, and loop-invariant computation information

- 1 Find all the basic i.v., by scanning the statements of L
- 2 Search for variables k , with a single assignment to k within L , having one of the following forms:
 $k := j * b$, $k := b * j$, $k := j / b$, $k := j \pm b$, $k := b \pm j$,
 $k := j * b \pm a$, $k := a \pm j * b$, where b is a constant and j is an i.v., basic or otherwise
 - (a) If j is basic, then for $k := j * b$, the triple for k is $(j, b, 0)$ (similarly for other forms)
 - (b) If j is not basic, then let its triple be (i, c, d) . We need to check two more conditions
 - (i) there is no assignment to j between the lone point of assignment to j in L and the assignment to k
 - (ii) no definition of j outside L reaches k

726 | 726 | Machine-Independent Optimizations

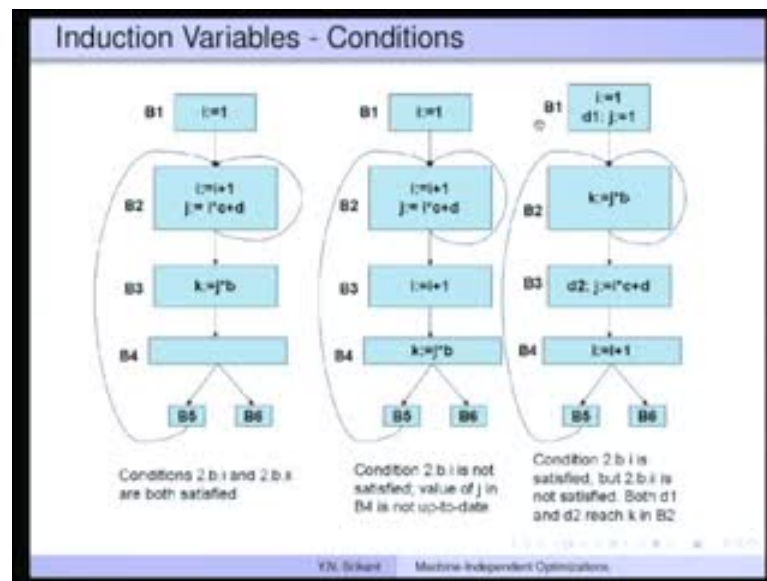
How do you detect induction variables? Basic induction variables are easy to find. You just scan the statements of the loop L whenever you have i equal to i plus minus n . They are basic induction variables. It is not necessary that basic induction variables are defined exactly once. They may be defined many times, but for derived induction variables in the family of a basic induction variable, the definition should be exactly once in the loop.

Search for variables such as k with a single assignment to k within the loop L having one of the following forms: There are many forms – j star b , b star j , etcetera. What is not permitted is b slash j because it becomes non-linear. Once we find something of this kind, we check for other conditions, which need to be satisfied. Just because k equal to b star j , it does not mean that k is an induction variable.

First, we check if j is basic. Let us assume that we are looking at k equal to j star b . If j is basic, then k equal to j star b and the triple for k is very simple, j , b and 0 . Similarly, for other forms as well; if it is j plus b , then we have j , 1 and b here (Refer Slide Time: 04:53). Similarly, for $(())$ Keep in mind that when you have (i,c,d) , it is i star c plus d and we just put those constants here. The conditions are necessary to be checked when j is not basic.

In other words, there is a basic induction variable i and then j is in the family of i . So, its triple is (i,c,d) . Now, we need to determine whether k , which is based on j can also be set to be in the family of i . What are the necessary conditions to be checked here? (Refer Slide Time: 05:32) Two conditions: First one is – there is no assignment to i between the lone point of assignment to j in L and the assignment to k ; Second condition is – no definition of j outside L reaches k .

(Refer Slide Time: 05:49)



Let us understand these conditions slightly better. Here is a basic induction variable i ; here is the definition of j and j is in the family of i ; here is the definition of k ; k equal to j star b and it is supposed to be in the family of j . Now, the conditions 2 b i and 2 b ii are satisfied here. Let us see what they are? (Refer Slide Time: 06:25) Condition (i) says – there is no assignment to i between the lone point of assignment to j in L and the assignment to k . Lone point of assignment to j and the assignment to k between these two (Refer Slide Time: 06:37), there is no assignment to i again. So, condition (i) is satisfied. No definition of j outside L reaches k . There is no definition of j outside L , which is reaching k . It is only this definition j , which is reaching this k (Refer Slide Time: 06:54). So, both the conditions are satisfied.

In this example, condition 2 b i is not satisfied. Let us see why. i is the basic induction variable as before, j is the derived induction variable in the family of i , then here (Refer Slide Time: 07:12) is the assignment to k , but there is an assignment to i as well. So,

between the assignment to j and assignment to k , there is an assignment to i . Now, the condition (i) is obviously violated. What is the problem? The problem is – j is updated here and assignment to i is **ok**. We now have j star b ; the old value of j is used here. So, there is no harm done. Whenever j is updated, the value of k is based on j and everything **ok**; it is not dependent on i at all.

Once we replace this assignment to k (Refer Slide Time: 07:58), which actually is a function of j by a function of i . Then, whenever i changes here, this k value will also change. This is the difficulty. Here, the old value of j is still available to me even though the value of i has changed, but once I replace this j (Refer Slide Time: 08:21); j star b by a suitable function of i because it is supposed to be in the family of i , then whenever i changes, k will also change. This will lead to a different program and it may be an erroneous condition. So, semantic violation has occurred here. Therefore, this is not a valid transformation. We cannot base any transformation on this in particular information. So, we cannot say that k is in the family of i .

In the last illustration here (Refer Slide Time: 09:02), condition 2 b i is satisfied. So, there is this assignment to k and this assignment to j , and there is no assignment to i in between. However, condition 2 b ii is violated. The reason is – for this particular k equal to j star b , there is a definition of j , which is reaching it from the outside; that is, from B1. There is another definition of j , which is reaching it from within the loop. So, there are two definitions of j . So, both B1 and B2 reach this particular use of (Refer Slide Time: 09:41) j . If this k , which is dependent on j is actually replaced by a function of i , this dependence goes away and it could be a mistake. There is a semantic violation if we replace this by a function of i . Therefore, this transformation is also not permitted. This is the reason why these conditions (Refer Slide Time: 10:09) have to be checked.

(Refer Slide Time: 10:12)

Detection of Induction Variables (2)

- If both j and k are temporaries in the same block, then checking the conditions (i) and (ii) above is easy
- Otherwise, we need to find all the basic blocks on the paths from the point of assignment to j , to the point of assignment to k , and check condition (i)
- Condition (ii) can be checked using u-d chain of j in the assignment to k
- Triple for k can be computed from (j, c, d) and the form of assignment to k
 - If $k := j + b$ and j is $i + c + d$,
 $k = (i + c + d) + b = (i + b + c) + (d + b)$
 - Hence the triple for k is $(i, b + c, d + b)$
 - Note that $b + c$ and $d + b$ are constants and can be evaluated by the compiler

YN Srinivas Machine-Independent C

How do we check these conditions? Fairly straight forward; If both j and k are temporaries in the same basic block, then checking conditions (i) and (ii) above is very easy. We do not have to go beyond basic blocks. So, we just check the space between the two definitions of j and k and see if there is a definition of i . There is nothing else, that is really going to cause problems. So, we are sure that j and k do not **live** beyond the basic blocks; that is the assumption here; j and k temporaries in the same block. There can be no definition of j , which arrives from outside the block. They live only within the block, they are defined, and they are actually not useful after the basic block. So, checking condition (i) is redundant, rather condition (ii) is redundant.

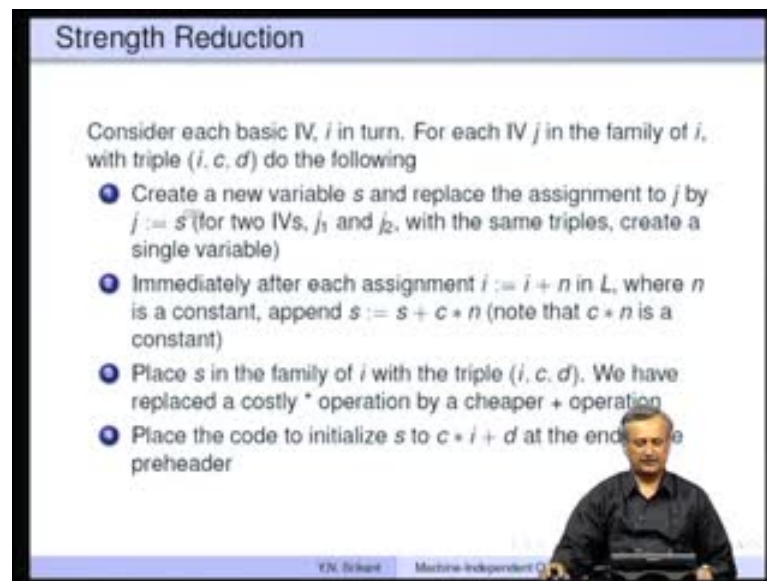
Otherwise, if they are not temporaries in the same basic blocks, then we need to find all the basic blocks on the paths from the point of assignment to j , to the point of assignment to k , and then check condition (i). So, you have to look at all the paths, find out all the basic blocks on the paths. So, every one of the paths must be taken. Only thing is that you do not have to go around the loops any number of times. So, you just have to go through the loop once and find all the basic blocks; that is it. Then, you can check condition (i) quite easily.

Condition (ii) can be very trivially checked using u-d chain of j in the assignment to k . So, this use-definition chain will tell us how many definitions of j are reaching that

particular point. If there is more than one, then we have a problem and immediately condition (ii) is violated.

Assuming that the conditions are satisfied, how do you compute the triple for k ? The triple for j is assumed to be (i, c, d) and let us say the form of assignment to k is $k = j * b$ and $j = i * c + d$. Make a simple substitution and find the constants. k becomes $i * c + d$ into b ; that becomes $i * b * c + d * b$ so this is our (Refer Slide Time: 12:51) c' and this is our b' . So, i, b' and $d * b$ are going to be our new triple. $b * c$ and $d * b$ are constants and therefore, can be evaluated by the compiler. That is how we get the new triple for k . If there is next induction variable dependent on k , we know how to convert it and make it dependent only on i .

(Refer Slide Time: 13:26)



Strength Reduction

Consider each basic block, i in turn. For each block j in the family of i , with triple (i, c, d) do the following

- 1 Create a new variable s and replace the assignment to j by $j := s$ (for two blocks, j_1 and j_2 , with the same triples, create a single variable)
- 2 Immediately after each assignment $i := i + n$ in L , where n is a constant, append $s := s + c * n$ (note that $c * n$ is a constant)
- 3 Place s in the family of i with the triple (i, c, d) . We have replaced a costly $*$ operation by a cheaper $+$ operation
- 4 Place the code to initialize s to $c * i + d$ at the end of the preheader

FN Srinani Machine Independent O

The next transformation that we need to understand is strength reduction. After this, we will look at induction variable elimination. Consider each basic induction variable, i . For each induction variable, j in the family of i ; let us say its triple (i, c, d) , we want to actually reduce the strength of j . If it is $4 * j$, we want to actually replace it with by some addition operation.

(Refer Slide Time: 14:14)

The slide is titled "Induction Variables - Strength Reduction Ex 1". It is divided into two columns. The left column is labeled "Before strength reduction for t5" and the right column is labeled "After strength reduction". Both columns show code for two loops, L1 and L2. In the "Before" column, the code for L1 includes `t5 = 4*i`. In the "After" column, this is replaced by `t7 = 4` and `t5 = t7`. The code for L2 remains the same in both columns. A person is visible in the bottom right corner of the slide, sitting at a desk.

```
Induction Variables - Strength Reduction Ex 1
```

Before strength reduction for t5	After strength reduction
<code>t1 = 202</code>	<code>t1 = 202</code>
<code>i = 1</code>	<code>i = 1</code>
<code>t3 = addr(a)</code>	<code>t3 = addr(a)</code>
<code>t4 = t3 - 4</code>	<code>t4 = t3 - 4</code>
L1: <code>t2 = i > 100</code>	L1: <code>t2 = i > 100</code>
<code>if t2 goto L2</code>	<code>if t2 goto L2</code>
<code>t1 = t1 - 2</code>	<code>t1 = t1 - 2</code>
<code>t5 = 4*i</code>	<code>t7 = 4</code>
<code>t6 = t4 + t5</code>	<code>t5 = t7</code>
<code>*t6 = t1</code>	<code>t6 = t4 + t5</code>
<code>i = i + 1</code>	<code>*t6 = t1</code>
<code>goto L1</code>	<code>i = i + 1</code>
L2:	<code>t7 = t7 + 4</code>
	<code>goto L1</code>
L2:	L2:

Before strength reduction for t5

After strength reduction

YN. Srikant Machine Independent C

To do that, we create a new variable s . Let me give you an example of that and then go back to the algorithm itself. Here, we have the basic induction variable, i , which is incremented; i equal to i plus 1, etcetera. $t5$ is the induction variable in the family of i ; it has 4 star i . Assuming that star is a costly operation and it needs to be replaced by a simpler plus operation; after transformation, it becomes $t5$ is equal to $t7$ and $t7$ equal to $t7$ plus 4 with an initialization of $t7$ equal to 4. So, there is no 4 star i any more. So, it has been replaced by an addition. That is why in the example (Refer Slide Time: 15:00) we had $t7$. Similarly, we have a new variable called s for each of the derived variables j , whose strength reduction has to be carried out; replace the assignment to j by j equal to s . We did that here (Refer Slide Time: 15:17). We replaced $t5$ is equal to 4 star i by $t5$ is equal to $t7$.

If there are two induction variables, j_1 and j_2 with the same triples (Refer Slide Time: 15:27), that means both of them advanced with the same constant value. Then, we can use the single variable for these two triples together.

Immediately after each assignment to i ; that is, i equal to i plus n in L , where n is a constant, place the assignment s equal to s plus c star n ; c star n is a constant obviously. this is the n (Refer Slide Time: 15:52) that we are talking about. Whenever i is incremented, we are also incrementing s by the appropriate amount, c star n because whenever i increments, j will also increment by the appropriate quantity. So, that is what

we want to do here; s is equal to s plus c star n . For example, j is equal to 4 star i , as i goes $1, 2, 3$, j goes $4, 8, 12$ and so on. That means addition of 4 . That is what we are doing here (Refer Slide Time: 16:28), or add c star n .

Here, (Refer Slide Time: 16:32) we have placed $t7$ equal to $t7$ plus 4 immediately after this statement i equal to i plus 1 because this constant is 4 and this is $(i,4,0)$. So, this is i equal to i plus 1 ; so, 4 star 1 is 4 ; so, $t7$ plus 4 .

(Refer Slide Time: 16:51)

Strength Reduction

Consider each basic IV, i in turn. For each IV j in the family of i , with triple (i, c, d) do the following

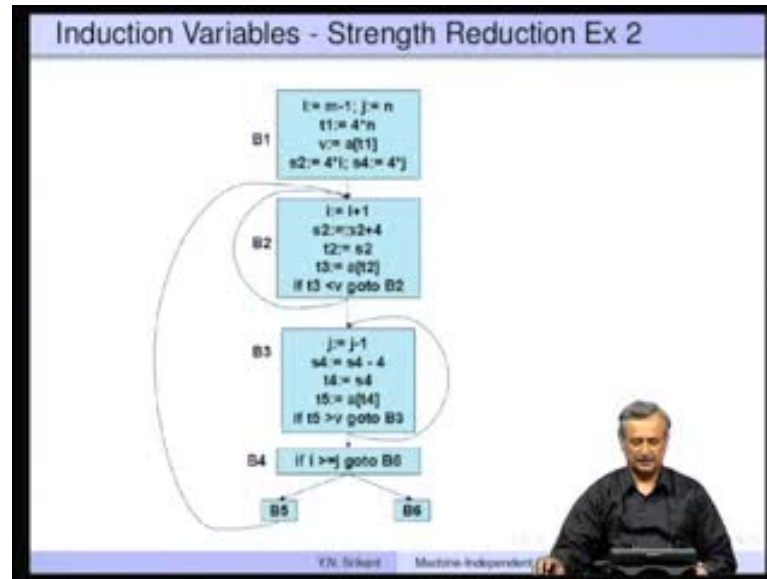
- 1 Create a new variable s and replace the assignment to j by $j := s$ (for two IVs, j_1 and j_2 , with the same triples, create a single variable)
- 2 Immediately after each assignment $i := i + n$ in L , where n is a constant, append $s := s + c * n$ (note that $c * n$ is a constant)
- 3 Place s in the family of i with the triple (i, c, d) . We have replaced a costly $*$ operation by a cheaper $+$ operation
- 4 Place the code to initialize s to $c * i + d$ at the end of the preheader

YN School Matteo Indipertini

Place s in the family of i with the triple (i,c,d) ; actually s is nothing but j . So, it is just a replacement for j . So, its triple is identical. We have replaced a costly star operation by a cheaper plus operation.

Place the code to initialize s to c star i plus d at the end of the preheader. We did that here (Refer Slide Time: 17:17); $t7$ equal to 4 . So, c star i plus d is 4 star i plus 0 ; that is, 4 . This is how the strength reduction has happened.

(Refer Slide Time: 17:32)



Let us take another simple example. This is example 2. For example, let us look at this (Refer Slide Time: 17:45). We had t_2 and t_4 , which are dependent on i and j respectively. Those have been replaced. Now, we have s_2 equal to s_2 plus 4 instead of $4 \star i$ and s_4 equal to s_4 minus 4 instead of $4 \star j$. The reason this has become (Refer Slide Time: 18:09) minus and this is still plus is this is i equal to i plus 1, whereas this is j equal to j minus 1. n is minus 1 here and that is why this has become minus 4. Then, there is initialization s_2 is equal to $4 \star i$ and s_4 is equal to $4 \star j$. **These two statements: s_2 is equal to and s_4 equal to; have in place immediately after i is equal to n ; j equal to...** So, this is a very simple strength reduction operation; star has been replaced by plus. This strength reduction happened within this loop and within this loop, respectively.

(Refer Slide Time: 18:47)

Elimination of Induction Variables

- Consider each basic IV i whose only uses are to compute other IV in its family and in conditional branches
- Consider j in i 's family with the triple (i, c, d)
- Replace $\text{if } i \text{ relop } x \text{ goto } B$ by the code sequence $\{r := c + x; r := r + d; \text{if } j \text{ relop } r \text{ goto } B\}$
- If c is negative, then we use relop in place of relop in the above code sequence
 - For example, if c is -4 , then $\text{if } i \geq x \text{ goto } B$ is replaced by the code sequence, $\{r := -4 + x; r := r + d; \text{if } j \leq r \text{ goto } B\}$
- Delete all assignments to the eliminated IV in loop
- Apply copy propagation (to eliminate statements)

YN Srikant Machine-Independent C

We did all these with a view to see if we can finally eliminate the induction variable. The idea is – in this case (Refer Slide Time: 19:00), here $t5$ increments by 4 in every loop, whereas i increments by 1. So, wherever we have the test for i , if we are able to replace it by the test for $t5$, then it is done. So, the first step in that process is to do strength reduction and then do induction variable elimination.

(Refer Slide Time: 19:30)

Induction Variable Elimination

```
t1 = 202
i = 1
t3 = addr(a)
t4 = t3 - 4
t7 = 4
L1: t2 = i > 100
if t2 goto L2
t1 = t1 - 2
t6 = t4 + t7
*t6 = t1
i = i + 1
t7 = t7 + 4
goto L1
L2:
```

Before induction variable elimination (i)

```
t1 = 202
t3 = addr(a)
t4 = t3 - 4
t7 = 4
L1: t2 = t7 > 400
if t2 goto L2
t1 = t1 - 2
t6 = t4 + t7
*t6 = t1
t7 = t7 + 4
goto L1
L2:
```

After eliminating i and replacing it with $t7$

YN Srikant Machine-Independent C

Let me give you the same example here. This is the old example. This $t7$ is equal to $t7$ plus 4 has replaced that $t5$ equal to 4 star i . So, we do not have that anymore. Now, i is

ready to be replaced by $t7$ itself. Instead of i greater than 100, can we put $t7$ greater than something? Yes, but $t7$ increments by 4 at a time. So, if i is compared with 100, $t7$ must be compared with a quantity, which is 4 times that. That is what we do here; $t7$ greater than 400 instead of i greater than 100. The rest of the loop remains the same and when we have got rid of that, this statement i equal to i plus 1 can be deleted.

How to do this systematically? (Refer Slide Time: 20:28) Consider each basic induction variable i whose only uses are to compute other variable in its family and in conditional branches. If this is so, we can eliminate this induction variable. Now, consider j in i 's family with the triple (i,c,d) . We want to remove i and replace it with it with j .

If there is a relational operation, rather conditional statement if i relop x goto B, we want to replace this i (Refer Slide Time: 21:06) by j , which is in the family of i . Then, what is the replacement for x ? If i becomes i star c , x should become x star c . If this becomes i star c plus d , this should become x star c plus d . So, i star c plus d is j and x star c plus d will be computed that will be put into the variable, r . That is why if i relop x goto B is the conditional statement, we replace it by r is equal to c star x , r equal to r plus d . So, that updates x appropriately and makes it get a value comparable to j . Then, we write if j relop r goto B. This is the transformation.

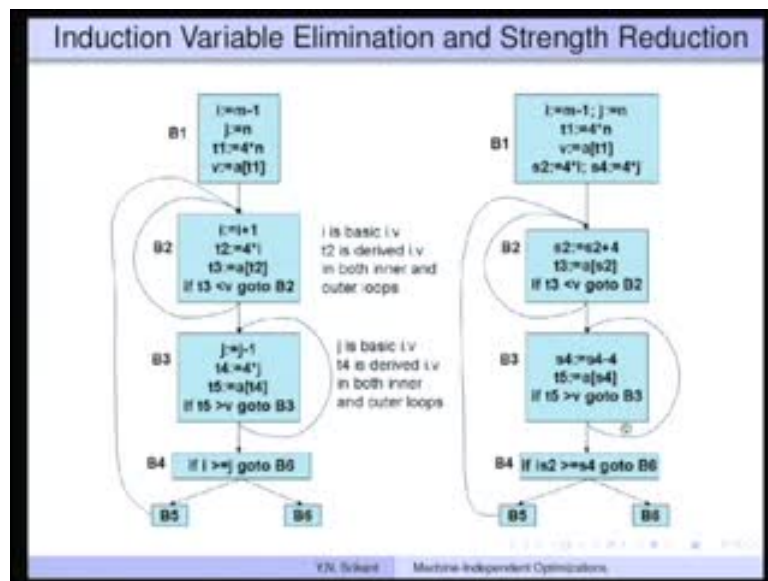
There is a small hitch here; what if c is negative? In other words, what if we have j equal to minus 4 star i ? i increases 1, 2, 3, but j will decrease in the negative direction minus 4, minus 8, minus 12, etcetera. In such a case, replacing this (Refer Slide Time: 22:27) i relop x by j relop r by this transformation would be become incorrect. Let us take a simple example: If you had minus 5 less than minus 2, it is a correct statement. Now, change it to plus 5; 5 less than 2 is obviously wrong. Similarly, if you had 5 greater than 2 and now, you change it to minus 5 greater than minus 2, that would be incorrect. What we need to do whenever we change the sign is – to complement the relational operator as well. If you have less than, you must replace it by greater than; if you had less than or equal to, then you must replace it by greater than or equal to and so on and so forth.

If c is negative, then we use relop bar; that is the complement of that relational operator in place of relop in the above code sequence. For example, if c is minus 4, then if i greater than equal to x goto B is replaced by the code sequence, r equal to minus 4 star x , r equal to r plus d , and then if j less than equal to r , goto B. So, this would be a correct

replacement for if i greater than equal to x . This is based on the assumption that we know the direction in which the decrement or increment is happening. **So, the sign of c .** That is why we always insist that c and d are constants. In theory, there could be loop-invariant values. The values comes from outside, but if we do not know the sign of that particular value, then this elimination of induction variables is not a possibility. We must know the direction in which the values increase or decrease; that is, the sign of this constant c ; otherwise, we cannot do anything about induction variable elimination.

We delete all the assignment to the eliminated induction variable in the loop; in this case, i (Refer Slide Time: 24:38). Apply copy propagation to eliminate statements of the form j equal to s , if necessary. Here (Refer Slide Time: 24:48), we did this replacement – instead of i greater than 100, it became $t7$ greater than 400. That is about it.

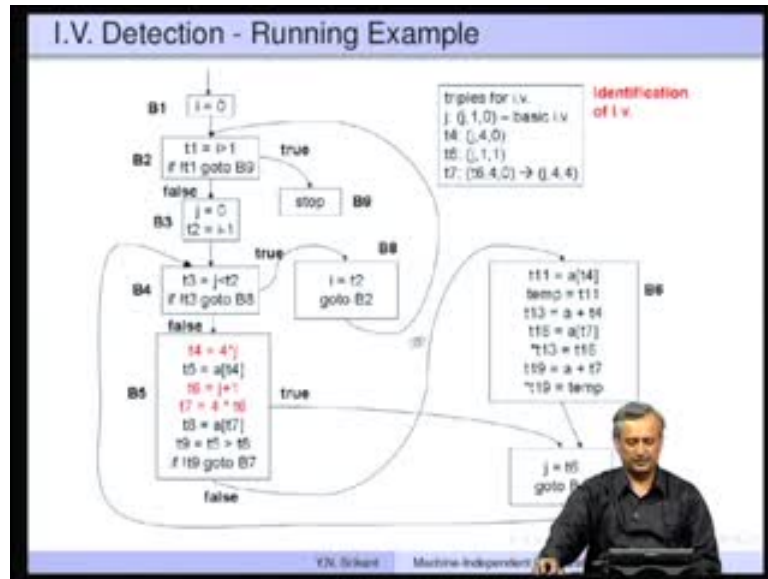
(Refer Slide Time: 24:58)



The second example; instead of i equal to i plus 1, $t2$ equal to 4 star i , we did strength reduction. It became $s3$ equal to $s2$ plus 4, but why do we have to keep i greater than equal to j here? j equal to j minus 1 and $t4$ equal to 4 star j are the other two statements. So, $t4$ equal to 4 star j has been replaced by $s4$ equal to $s4$ minus 4.

Can we replace i greater than equal to j by $s2$ and $s4$, respectively? It so happens that we can; it is just that it should be the inverse of the condition. So, $s2$ not of $s2$ greater than equal to $s4$. If we have that, **then goto B6; works perfectly here.** So, this is the way we actually do induction variable elimination.

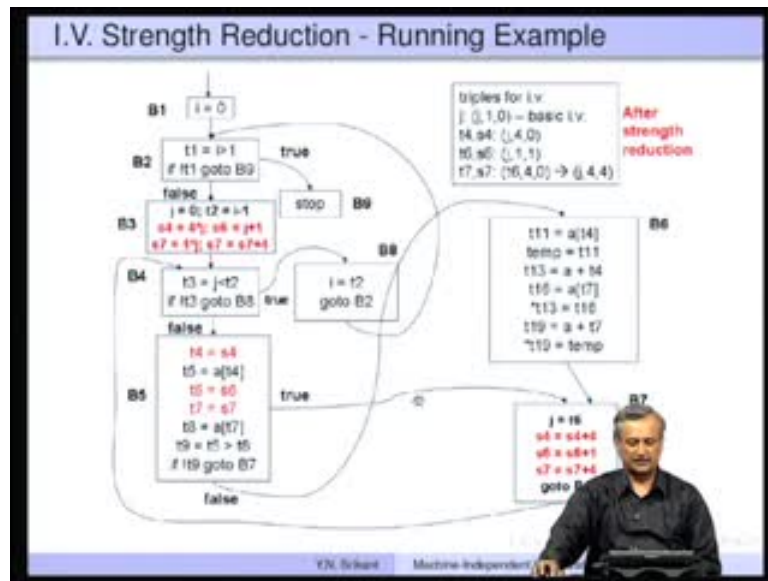
(Refer Slide Time: 25:59)



Let us take a running example starting from i ; looking at the old program that we had. We really have several induction variables here; they are all listed here. i is also one of them, but let us ignore i . Let us look at j only and its derivatives. j is a basic induction variable. So, $(j, 1, 0)$ is its triple. Then, we have $t4$ equal to $4 * j$ here. So, that is a derived induction variable in the family of j with $(j, 4, 0)$. Then, $t6$ is j plus 1. So, that is another derived induction variable with the triple $(j, 1, 1)$.

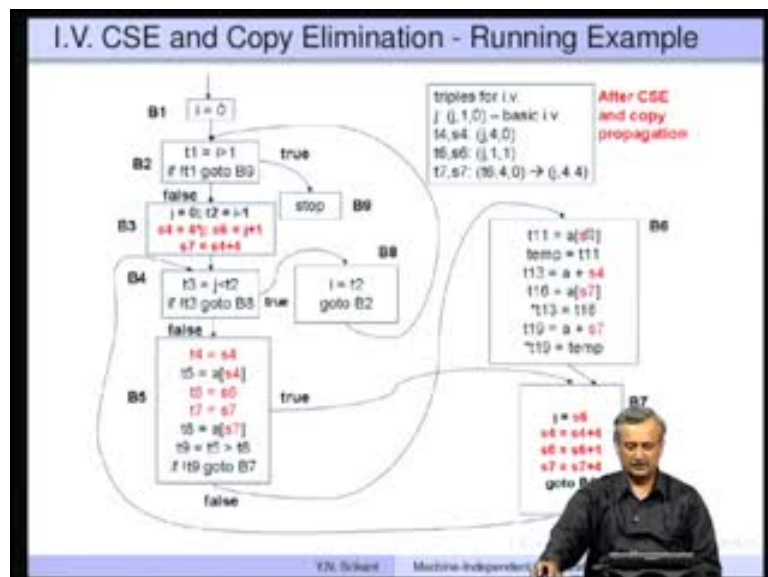
Then, $t7$ is one more says (Refer Slide Time: 26:50) – $4 * t6$. So, it is in the family of $t6$; so, $(t6, 4, 0)$, but $t6$ itself happens to be in the family j . So, $t7$ is also in the family of j with the triple $(j, 4, 4)$. These are the induction variables that we have deduced in this loop.

(Refer Slide Time: 27:13)



Now, let us do strength reduction. Here, (Refer Slide Time: 27:19) t_4 equal to $4 * j$ was replaced by t_4 equal to s_4 and s_4 equal to $s_4 + 4$. Then, t_6 equal to $j + 1$ was replaced by t_6 equal to s_6 and s_6 is equal to $s_6 + 1$. t_7 equal to $4 * t_6$ was replaced by t_7 is equal to s_7 and s_7 equal to $s_7 + 4$. Then, appropriate initializations were all carried out in this particular preheader. So, this is strength reduction.

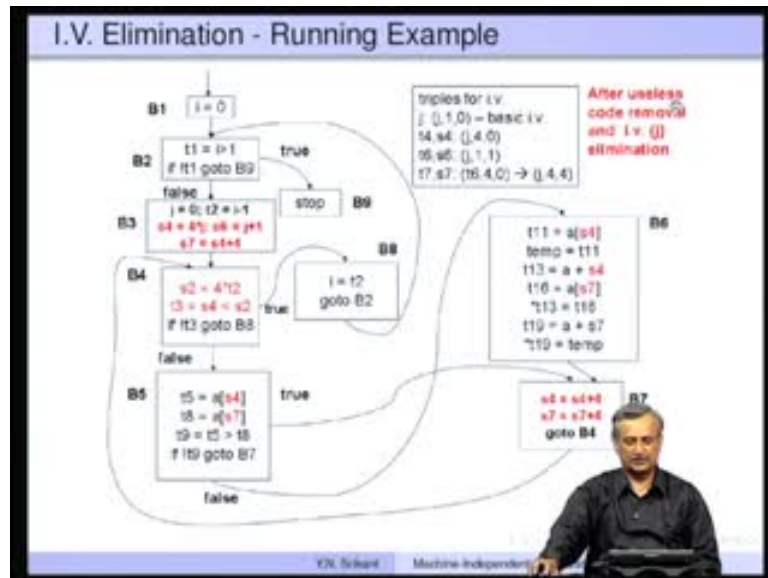
(Refer Slide Time: 27:52)



After common sub-expression elimination and copy propagation. For example, (Refer Slide Time: 28:00) all these t_4 etcetera becomes s_4 . So, these are the copy propagations

that have taken place; a of s4, a plus s4, a of s7, a plus s7 etcetera. After we did this, where was... Here, (Refer Slide Time: 28:24) we had 4 star j, 4 star j. So, these are two common sub-expressions. Those were also removed and that is why CSE is mentioned here (Refer Slide Time: 28:34); little more efficient code.

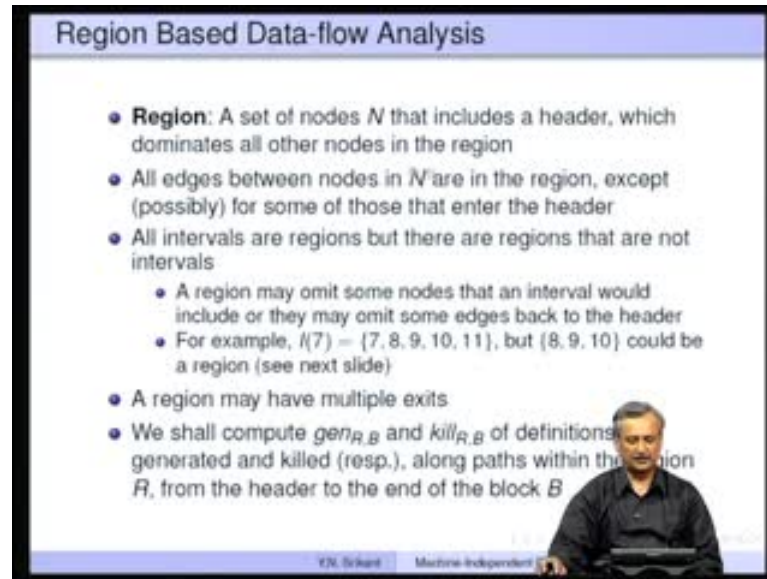
(Refer Slide Time: 28:38)



Finally, removed the useless code that is corresponding to the i and did induction variable j elimination. We had in this; for example, (Refer Slide Time: 28:51) we had – if t3 equal to j less than t2; if not, t3 goto B3 or B8. That was replaced by t2. So, s2 became 4 star t2 and t3 equal to s4 less than s2. So, we replaced for example, (Refer Slide Time: 29:09) j by s4 and t2 was replaced by its appropriate multiplicand; that is, s2; 4 star t2. So, that is how the induction variable elimination took place. Now, this becomes not of t3; it is not dependent on j anymore.

The quadruples for j were all removed and only those (Refer Slide Time: 29:34), which are required – s4, s7, etcetera were retained. This is a detailed example of how induction variable detection, strength reduction and induction variable elimination happens for a nontrivial program.

(Refer Slide Time: 29:50)



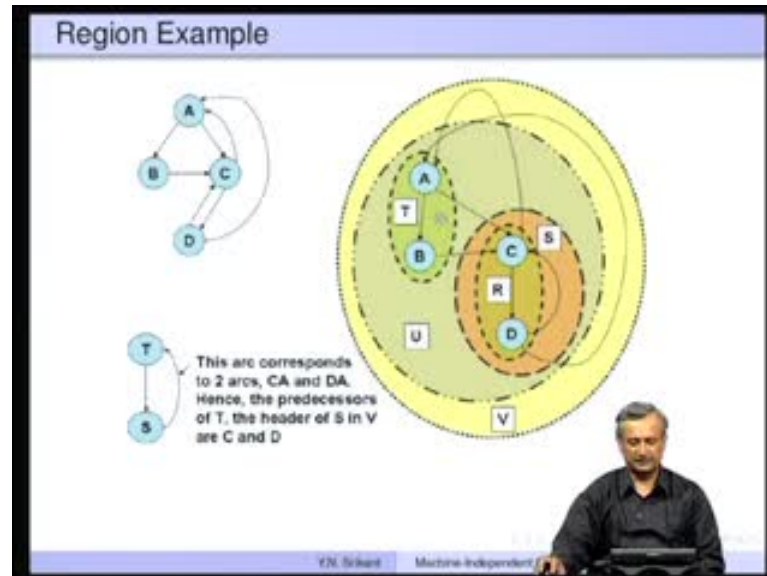
Region Based Data-flow Analysis

- **Region:** A set of nodes N that includes a header, which dominates all other nodes in the region
- All edges between nodes in N are in the region, except (possibly) for some of those that enter the header
- All intervals are regions but there are regions that are not intervals
 - A region may omit some nodes that an interval would include or they may omit some edges back to the header
 - For example, $I(7) = \{7, 8, 9, 10, 11\}$, but $\{8, 9, 10\}$ could be a region (see next slide)
- A region may have multiple exits
- We shall compute $gen_{R,B}$ and $kill_{R,B}$ of definitions generated and killed (resp.), along paths within the region R , from the header to the end of the block B

YN. Srikant Machine-Independent

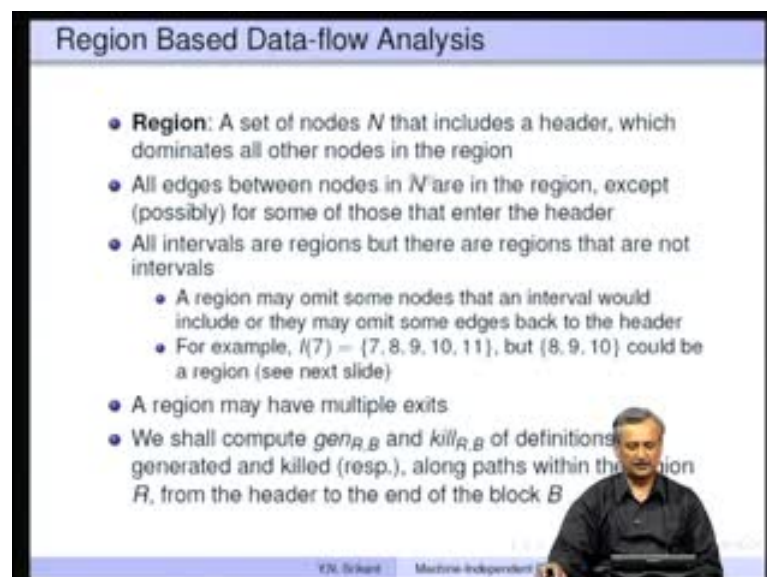
So, that is as far as some of the basic optimizations are concerned. Let us look at a very interesting version of data-flow analysis called region based data-flow analysis. In one of the previous lectures, we talked about iterative data-flow analysis in which the basic blocks for all traverse in particular order to find a solution for the data-flow analysis problem. Whereas here, we divide the control flow graph into regions; we looked at regions very briefly. Let us recapitulate - what exactly regions are. So, region is a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except; possibly for some of those that enter the header.

(Refer Slide Time: 31:05)



It is very easy to give you some examples of this. Let me go little further; Here, for example, if this is the big region. When we are constructing the yellow region outside this, the arcs A C and A D are not in the region U, they are in the region V. So, when we consider region U, we do not consider these two arcs; we will consider it only when we go to region V.

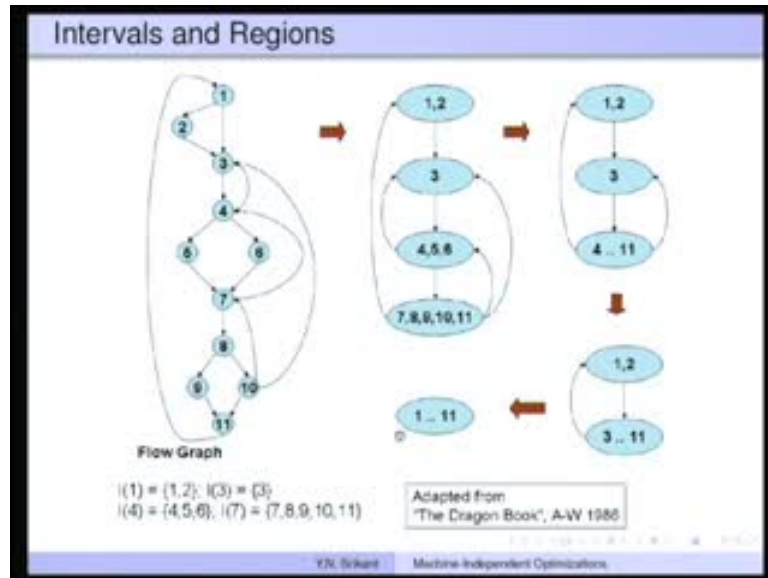
(Refer Slide Time: 31:34)



All intervals are regions, but there are regions that are not intervals. A region may omit some nodes that an interval would actually include or they may omit some edges also

back to the header. For example, the interval for 7 was 7, 8, 9, 10, 11, but 8, 9, 10 could be a region.

(Refer Slide Time: 31:55)

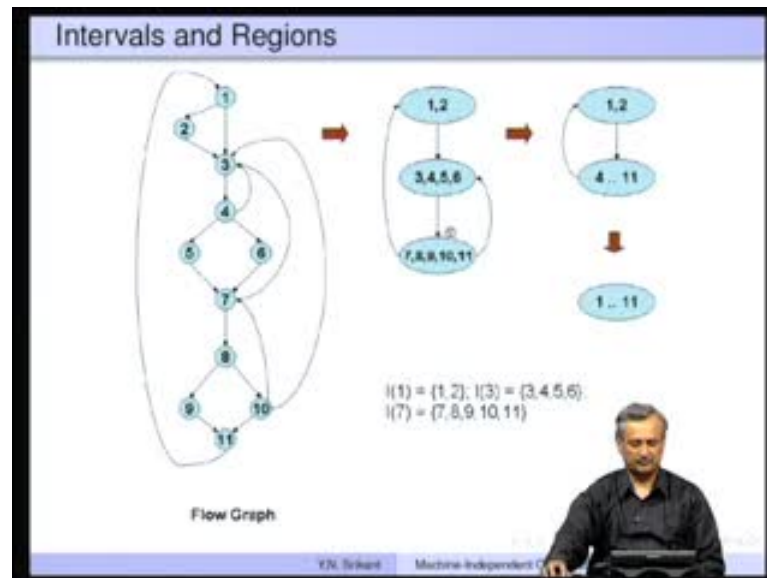


Let me show you that example. So, **i 7** is 7, 8, 9, 10, 11; this entire thing, but 8 9 10 could be just this region. This could be a region on its own; 8 **nominates** 9 and 10. So, that is how this becomes a small region on its own.

A region may have multiple exits. This region – 8, 9, 10 has 2 exits; so, no problem. Now, what we are going to do is (Refer Slide Time: 32:30 – we will compute gen of R comma B and kill of R comma B of definitions generated and killed along the paths within the region R, from the header to the end of the block B. This will become clear as we go along.

This is the interval graph (Refer Slide Time: 32:55) that we had already looked at. This is the interval 1 2 3 and then, this reduces further, so on and so forth.

(Refer Slide Time: 33:04)



This was the other interval graph. Interestingly, if you look at 3 as the header, then this 3 4 5... There is a minor change here in the graph from... Instead of this (Refer Slide Time: 33:18), we made it that. So, this changes the intervals structure; so, 3 4 5 6 become an interval and rest of its 7 8 9 10 11 become another intervals. Finally, reduction takes place.

Each of these (Refer Slide Time: 33:32) could be a region actually; so, no problem; an interval could be a region. This is an example of how regions are, or each of these (Refer Slide Time: 33:42) could be a region. So, there are 4 regions in this example and 3 regions in this example.

(Refer Slide Time: 33:50)

Region Based Data-flow Analysis (2)

- These will be used to define a transfer function $trans_{R,B}(S)$, that tells for any set S of definitions, what subset of definitions reach the end of B by travelling along paths wholly within R , assuming that all and only the definitions in S reach the header of R
- $trans_{R,B}(S) = gen_{R,B} \cup (S - kill_{R,B})$
- $trans_{U,B}(\phi) = OUT[B] = gen_{U,B}$, where U is the region consisting of the entire flow graph
- We need to provide a method to compute the transfer functions $trans_{R,B}$, for progressively larger regions defined by some $(T_1 - T_2)$ transformation of a CFG
- Since $OUT[B] = gen_{U,B}$, we need to compute only $gen_{R,B}$ and $kill_{R,B}$, for each basic block, for progressively larger regions
- Interestingly, this approach does not compute $IN[B]$ at all

Y.N. Srikant Machine-Independent Optimizations

What exactly is region based data-flow analysis? The basic idea is to define a transfer function called $trans$ of R comma B with a parameter S . So, this will tell us that for the input S , which is a set of definitions; Let us say - we are looking at reaching definition problems. For any set S , which is sent as a parameter; that is a set of definition in this example, what subset of definitions reach the end of the basic block B by travelling along paths wholly within R ; this is important; we are not looking at edges, which are not in the region, assuming that all and only definitions in S reach the header of R . So, there is no other way of reaching the header of R ; only the definitions in S will reach the header of R . Then, how do they travel inside the region R is what is captured by this **trans** function.

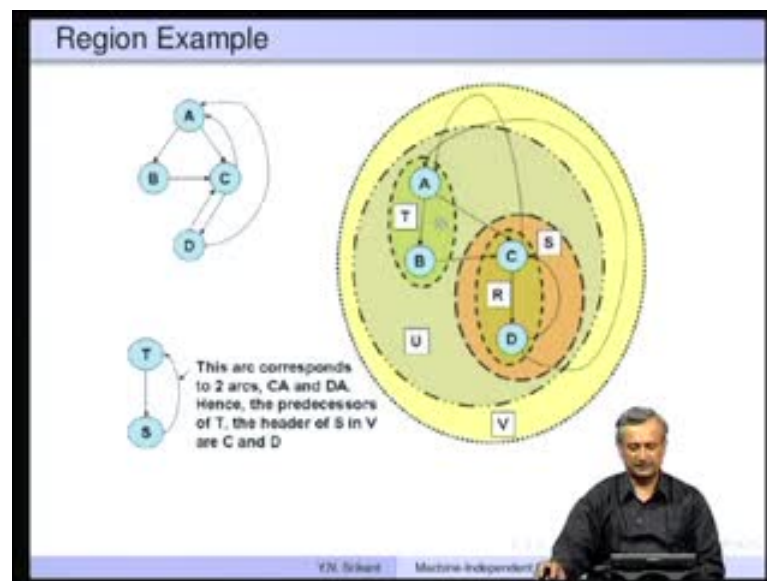
$trans$ of R comma B with the parameter S would be whatever is generated within the region R ; that is, gen of R comma B for that basic block B and reach the end of the basic block B , union S minus whatever is killed by the basic block B with respect to the region R . So, if you look at the entire region; let us say - $trans$ of U comma B and then let us say - the incoming set of definitions is ϕ ; that is what the initialization we always had. That becomes out B . This is the (Refer Slide Time: 35:43) out function that we defined in iterative data-flow analysis. So, here is the connection between the region based data-flow analysis in which we want to find $trans$ functions and the out functions that we have in iterative data-flow analysis. This out B is nothing but gen of U comma B because S is ϕ here; ϕ minus something is ϕ . So, gen of R comma B becomes gen

of $U \cup B$ with R equal to U , where U is the region consisting of the entire flow graph.

Basically, we need to provide a method to compute the transfer function, $trans$ of $R \cup B$ for progressively larger regions defined by some $T_1 T_2$ transformation of a control flow graph. You could use interval methods and define regions, but in our example we are using $T_1 T_2$ transformation to define regions.

Since out_B is gen of $U \cup B$, we need to compute only gen of $R \cup B$ and $kill$ of $R \cup B$ for the basic blocks. These are needed for computing gen for progressively larger regions. We do not have to compute in_B function at all; it is not necessary in this approach.

(Refer Slide Time: 37:03)

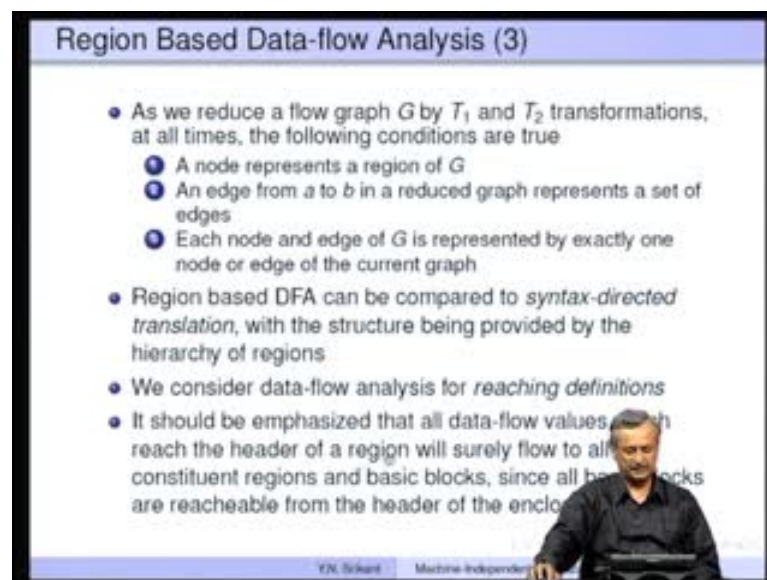


There are couple of observations here. Before we take up an example, let me show you this structure (Refer Slide Time: 37:05). Here, is our control flow graph for which are going to do region based data-flow analysis. At the inner most level are the regions A, B, C, D. When we combine them using T_2 transformation, we get regions T and R. This edge D to C is outside the region R. When we apply the T_1 transformation, we get on this region R; we get region S.

Finally, when we apply T_2 transformation between T and S , we get the region U . The edges from D to A and C to A are not part of U , but they are part of V . So, we need to apply T_1 transformation on U in order to get V and V is the final.

We are going to start our region based data-flow analysis from the basic blocks and grow the regions. This error key of regions is provided by the $T_1 T_2$ analysis. As we perform the transformations of T_1 and T_2 , we keep track of the regions as well. We compute the gen information for the basic blocks and then provide a method to compute the gen information for T , R , S , U , V , etcetera when we perform either the T_1 or T_2 transformations.

(Refer Slide Time: 38:33)



Region Based Data-flow Analysis (3)

- As we reduce a flow graph G by T_1 and T_2 transformations, at all times, the following conditions are true
 - 1 A node represents a region of G
 - 2 An edge from a to b in a reduced graph represents a set of edges
 - 3 Each node and edge of G is represented by exactly one node or edge of the current graph
- Region based DFA can be compared to *syntax-directed translation*, with the structure being provided by the hierarchy of regions
- We consider data-flow analysis for *reaching definitions*
- It should be emphasized that all data-flow values which reach the header of a region will surely flow to all constituent regions and basic blocks, since all basic blocks are reachable from the header of the enclosing region.

YN Srikant Machine-Independent

As we reduce the flow graph G by T_1 and T_2 transformations, the following conditions are true at all times. A node represents a region of G . An edge from a to b in a reduced graph represents a set of edges. So, this is clear. For example, (Refer Slide Time: 39:00) this edge S to T ; so, we have S here and this is T ; this is something that I already told you. These are the two edges corresponding to one edge in the bigger reduced graph.

Each node and edge of G is represented by exactly one node or edge of the current graph. In the reduced graph, there is representation; each node and edge of G will be represented by exactly one node or region. So, you cannot have a node of this graph G (Refer Slide Time: 39:40) in say A ; it cannot be present in more than one region at any

point in time; it can be a part of exactly one region. The same is true for an edge as well; an edge cannot belong to two regions at any point in time.

Region based DFA can be compared to syntax-directed translation, with the structure being provided by the hierarchy of regions. In other words, let us say that we are trying to generate machine independent code; quadruples for expressions. We start from the lowest level, look at the expression, and then if it is a star b plus c, we generate code for a star b, put it into a temporary, then use that t plus b, generate code t plus b, so on and so forth. So, this is the syntax-directed translation. Similar thing is done in the case of region based analysis also (Refer Slide Time: 40:51). The regions grow bigger and bigger just like the syntax tree becomes bigger and bigger. From inside that is, corresponding to leaves; that is, the basic region here, we go to the outer most region; that is the route of the syntax tree, we actually compute the data flow information.

We consider the data-flow analysis for reaching definitions in this particular case and immediately afterwards, we will see how to do available expression analysis as well. It should be emphasized that all data-flow values, which reach the header of a region will surely flow to all the constituent regions and basic blocks since all the basic blocks are reachable from the header of the enclosing region. So, what we are trying to say here is – (Refer Slide Time: 41:50) in each region, there is full connectivity. So, the header of any region can reach any node in that particular region; there is connectivity. So, if some information reaches the header of a particular region, then it will definitely flow to every node in that particular region; the smallest node as well. That is what we say here. This connectivity information is very important because the only way we can enter a region is through the header and if the header is not connected to some node, then some information cannot flow. So, our basic assumption is that; you can reach every node in that particular region. However, the region formation itself ensures this; so, there is nothing to worry.

(Refer Slide Time: 42:41)

The slide is titled "Region Building by T2 Trans. - Reaching Def". It features a control flow graph on the left with three regions: R1 (top), R2 (bottom right), and R (the union of R1 and R2). Each region contains a basic block B. Arrows indicate control flow from R1 to R2 and from R2 back to R1. A text box at the bottom left says "For reaching definitions problem".

Basic regions:
 $gen_{R,B} = gen[B]$
 $kill_{R,B} = kill[B]$

Region building by T2

For basic blocks B within R1,
 $gen_{R1,B} = gen_{R1,B}$
 $kill_{R1,B} = kill_{R1,B}$

Edges from R2 to header of R1 are not part of R

For basic blocks B within R2,
 $gen_{R2,B} = gen_{R2,B} \cup (G - kill_{R2,B})$
 $kill_{R2,B} = kill_{R2,B} \cup (K - gen_{R2,B})$
where $G = \bigcup gen_{R1,P}$ and $K = \bigcap kill_{R1,P}$
for all predecessors P of R2 in R1

At the bottom right, a small video inset shows a man sitting at a desk.

There are two transformations: T 1 and T 2. For each of these transformations, we provide the method of computing the transfer functions. As I told you, we need to compute only the gen and kill information for each basic block corresponding to each region; from the lowest level to the highest level.

For the basic regions; that is, the basic blocks, we know how to compute gen B and kill B is another quantity, which we already know how to compute from iterative data-flow analysis. So, **gen B of B comma B** instead of gen R comma B; that is, the lowest level. gen of B comma B is nothing but gen B and kill of B comma B is nothing but kill B.

With this information, let us take a region such as R, which is built by a T 2 transformation from the regions R1 and R2. So, R1 has been built before, R2 was also built before. Now, there is an edge between R1 and R2. So, we can collapse R1 and R2 and make it a single region R. So, this was the transformation if you have supplied.

Now, the structure is available to us. How exactly the transformation happened? Given the transfer functions gen and kill for R1 and R2, how do we compute the transfer function gen and kill for the region R for the same basic block B in these two (Refer Slide Time: 44:27?)

Region building by T2; suppose you consider the basic block within the region R1 such as B. There are many basic blocks, but we have shown only one. R2 does not affect R1

because there is no path to go back to R1 within this region R (Refer Slide Time: 44:46). You cannot go from R2 to R1 within this region R. The only connection between regions R1 and R2 is this particular edge, which may correspond to more than one edge in the actual control flow graph, but there is no back edge; nothing; you cannot go back. That is the way the regions have been constructed. Therefore, since R2 does not affect R1 at all, the gen of R comma B; that is, **gen of** with respect to region R for the basic block B, is same as gen of this particular B with respect to R1; that is, R1 comma B. Similarly, kill information is also the same. Kill of R comma B is same as kill of R1 comma B.

Now, this is what i was saying (Refer Slide Time: 45:34) – Edges from R2 to R1 are not part of region R. So, we do not have to worry about going from here to here. If you look at basic block within the region R2, they are definitely affected by the region R1, but which blocks in R1 affect R2? Since the connection is just this, the only block, which can affect R2. Or, if this edge corresponds to let us say – three edges, there will be three blocks connected to those three edges here. So, those are the only three edges, which will hurt R2. Let us take this example (Refer Slide Time: 46:19). Here, T and S; this C is the header of the region S. So, there is an edge from T to C and A to C; this is the region. T corresponds to R1 and S corresponds to R2. So, these are the only two basic blocks, rather regions from which we can enter C.

Therefore, we need to consider blocks corresponding to the predecessor of the header of R2. Here, for example (Refer Slide Time: 46:58), this is a header of R2. We want to look at the predecessor of the header of R2. Those are the two blocks – A and B. So, gen of R comma B now (Refer Slide Time: 47:12) will be gen of R2 comma B, whatever is generated by V with respect to the region R2 union G minus kill of R2 comma B. What is G? G is union of gen of R1 comma P for all predecessors P of the header of R2 in R1. So, those are the blocks I was talking about. Header of R2 is here; there are many predecessors. Take all those predecessors; they are only ones, which can hurt R2. Take the union of the gen of all those predecessors; they will all be able to reach the header of R2. That is the set G. From that set G, remove whatever is killed by this R2. So, kill of R2 comma B; killed by B with respect to R2. Add gen of R2 comma B; generated by B with respect to R2. Then, you get whatever is generated by B with respect to the region R. So, this should be clear.

Kill is very similar. Kill of R2 comma B union K minus gen of R2 comma B. So, $(\cap) K$ is intersection of kill of R1 comma P. Why? If there are many predecessors like this (Refer Slide Time: 48:34), unless the definition is killed along both these paths, we do not take it as killed. That is the reason why we take the intersection. For generation, it can reach along any one of the paths, but for killing, it should be killed along all paths. That is why this is an intersection and this is a union (Refer Slide Time: 48:53).

(Refer Slide Time: 48:55)

Region Building by T1 Trans. - Reaching Def

For reaching definitions problem

Region building by T1

$$\text{gen}_{R_1,B} = \text{gen}_{R_1,B} \cup (G - \text{kill}_{R_1,B})$$

$$\text{kill}_{R_1,B} = \text{kill}_{R_1,B}$$

where, $G = \bigcup \text{gen}_{P,R_1}$ for all predecessors P of the header of R1 in R

It is not necessary to compute $\text{kill}_{R_1,B}$ as in the previous case (T2)

A definition gets killed going from the header to B if it is killed all acyclic paths, and hence edges incorporated into R will cause more definitions to be killed.

T.N. Saha, Madhav Indraprastha

What about the transformation T 1? T 1 is like this. There is a region R1, there is a self-loop and we want to get rid of it. So, that is the transformation T 1. If you look at the kill set for the region R, it is same as the kill set of the basic block B with respect to R1. Why? A definition gets killed going from the header to B, if and only if it is killed along all acyclic paths. Hence, back edges such as this incorporated into R will not cause more definitions to be killed. So, from the header here, along every path to B, the definitions are killed. So, those are the kill definitions that we already know.

This back edge will not create further problems and therefore, kill of R comma B is nothing but kill of R1 comma B. So, whatever gets killed from the header to B, is kill of R1 comma B and that is precisely, kill of R comma B as well; nothing more can be added by this. Whatever is killed along all paths are already taken. So, back edge will not add extra information.

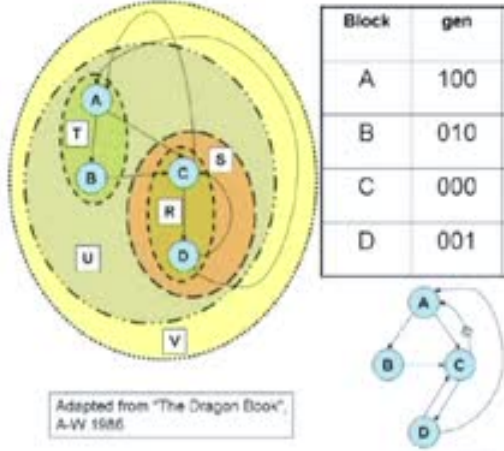
The same is not true for the gen set. There may be $\text{gen of } R1 \text{ comma } B \text{ union } G \text{ minus kill of } R1 \text{ comma } B$ is $\text{gen of } R \text{ comma } B$. Why? G is union of $\text{gen of } R1 \text{ comma } P$, for all the predecessors. So, it is possible that some definitions here (Refer Slide Time: 50:43) come... We are looking at all possible paths and any one of these paths can generate a definition; it is unlike kill. So, maybe from here, I go back and then take some paths, which come here; that could be generating a definition for us.

Whereas, in the case of kill, every path was covered. Here, (Refer Slide Time: 51:11) not all paths are covered; any one of the paths is good enough for our generation. Therefore, we need to look at all the predecessors of this particular edge. This is a compound edge. So, there may be many predecessors corresponding to this compound edge. Look at all of them and then take... To give you an example, here, (Refer Slide Time: 51:34) in this region V , we had a self-loop. So, these two edges corresponded to a single compound edge. For this header, we had two predecessors C and D ; one was this (Refer Slide Time: 51:50) and one was that. Both these together corresponded to a compound edge. This compound edge (Refer Slide Time: 51:56).

We take the predecessors of this compound edge. When we actually travel along this edge, they may actually reach the header of $R1$ and then further reach B . That is the reason why we take the union of all these gen sets of the predecessors, then remove whatever is killed by this basic block, and add whatever is generated by this basic block. That gives us $\text{gen } R \text{ comma } B$. So, this is very similar to what we did in the T_2 transformation, but this is different because of this particular (Refer Slide Time: 52:37) reason. This is very similar. So, we take the predecessors and then they can all come here; possibly along any one of the paths.

(Refer Slide Time: 52:49)

Region Based RD Analysis - An Example (1)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000


Adapted from "The Dragon Book", A-W 1986

Y.N. Srikant Machine-Independent Optimizations

Here, is an example, region based analysis. Here, we have all the regions that I showed you. Let us say – there are 4 blocks, here are the gen and kills sets, and here is our control flow graph.

(Refer Slide Time: 53:10)

Region Based RD Analysis - An Example (2)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A-W 1986

- Building region R from regions C and D by T2 transf.
- $gen_{R,C} = gen_{C,C} = 000$; $kill_{R,C} = kill_{C,C} = 010$
- Header of D is D and pred. of D in C is C
- $G = gen_{C,C} = 000$ and $K = kill_{C,C} = 010$
- $gen_{R,D} = gen_{D,D} \cup (G - kill_{D,D}) = 001 + (000 - 000) = 001$
 $kill_{R,D} = kill_{D,D} \cup (K - gen_{D,D}) = 000 + (010 - 001) = 010$

Y.N. Srikant Machine-Independent Optimizations

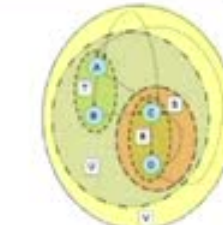
For example, if you are building region R from C and D by T 2 transformation, gen of R comma C is gen of C comma C. This is at the lowest level. So, it is same as the gen and kill sets; kill is similar.

The header of D is just D and predecessor of D in C is just the node C. This is a T 2 transformation; we are computing G and K. We are looking at the predecessors; that is only one, which is C. So, we get gen C comma C. This also becomes only C; kill of C comma C.

Now for the bigger region R (Refer Slide Time: 53:57), gen of R comma D from the equation is gen of D comma D union G minus kill of D comma D; that gives us 0 0 1. Similarly, kill of R comma D is kill of D comma D union K minus gen of D comma D; that gives us 0 1 0.

(Refer Slide Time: 54:15)

Region Based RD Analysis - An Example (3)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A. W. 1989

- Building region S from region R by T1 transformation
- The only predecessor of the header C, within S is D
- Therefore, $G = gen_{R,D} = 001$
- $kill_{S,C} = kill_{R,C} = 010$; $kill_{S,D} = kill_{R,D} = 010$
- $gen_{S,C} = gen_{R,C} \cup (G - kill_{R,C}) = 000 + (001 - 010) = 001$
- $gen_{S,D} = gen_{R,D} \cup (G - kill_{R,D}) = 001 + (001 - 010) = 001$


Y.N. Srikant Machine-Independent Optimizations

Now, we apply the T 1 transformation to get region S from R. So, there is only one predecessor for this header node C, which is D within S. This is the edge we are looking at; this is D. G is gen of R comma D; R is this region. So, 0 0 1 as we computed before. gen of R comma D is 0 0 1.

The kill set is not modified; it is same as that of R comma C. That is written here. The gen set is computed using that formula gen union G minus kill. So, we get the appropriate 0 0 1 and 0 0 1 for the C and D nodes with respect to the region S.

(Refer Slide Time: 55:00)

Region Based RD Analysis - An Example (4)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A.9, 1988


- Building region T from regions A and B by T2 transf.
- $gen_{T,A} = gen_{A,A} = 100$; $kill_{T,A} = kill_{A,A} = 010$
- Header of B is B and pred. of B in A is A
- $G = gen_{A,A} = 100$ and $K = kill_{A,A} = 010$
- $gen_{T,B} = gen_{B,B} \cup (G - kill_{B,B}) = 010 + (100 - 10) = 1010$
 $kill_{T,B} = kill_{B,B} \cup (K - gen_{B,B}) = 101 + (010)$

Y.N. Srikant Machine-Independent

Then, we have building region T; that is, this which is very similar to building region R. So, let us not spend too much time on it. Similarly, we get gen of T comma A, kill of T comma A, G, K, gen of T comma B and T comma B. This is very similar to what we did for C and D to build region R.

(Refer Slide Time: 55:25)

Region Based RD Analysis - An Example (5)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A.9, 1988


- Building region U from regions T and S by T2 transf.
- $gen_{U,A} = gen_{T,A} = 100$; $kill_{U,A} = kill_{T,A} = 010$
- $gen_{U,B} = gen_{T,B} = 010$; $kill_{U,B} = kill_{T,B} = 101$

Y.N. Srikant Machine-Independent

Then, we want to build the region U from T and S; this is by T 2 transformation. These two: A and B will not be modified too much because this corresponds to region R1. So, they are as it is; they are as they are.

(Refer Slide Time: 55:44)

Region Based RD Analysis - An Example (6)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book" Aug 1985


- Building region U from regions T and S by T2 transf.
- Header of S is C and pred. of C in T are A and B
- $G = gen_{T,A} \cup gen_{T,B} = 110$ and
 $K = kill_{T,A} \cap kill_{T,B} = 000$
- $gen_{U,C} = gen_{S,C} \cup (G - kill_{S,C}) = 001 + (110 - 010) = 101$
 $kill_{U,C} = kill_{S,C} \cup (K - gen_{S,C}) = 010 + (000 - 000) = 010$
 $gen_{U,D} = gen_{S,D} \cup (G - kill_{S,D}) = 001 + (110 - 000) = 111$
 $kill_{U,D} = kill_{S,D} \cup (K - gen_{S,D}) = 010 + (000 - 001) = 001$

Y.N. Srikant Machine Independent CS & Engg.

Whereas, for this region S, it is different; so, the header of S is C and predecessors of C are B and A. So, there are two nodes here. Because of that, we have to compute G and K with these two predecessors: gen union gen; A and B. kill is intersection; A and B. So, we get this. Then, apply the gen equal to gen union G minus kill formula and obtain these four quantities.

(Refer Slide Time: 56:19)

Region Based RD Analysis - An Example (8)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book" Aug 1985

- Building region V from region U by T1 transf.
- Header of U is A and pred. of A in U are C and D
- $G = gen_{U,C} \cup gen_{U,D} = 101$
- $gen_{V,A} = gen_{U,A} \cup (G - kill_{U,A}) = 100 + (101 - 010) = 101$
 $gen_{V,B} = gen_{U,B} \cup (G - kill_{U,B}) = 010 + (101 - 101) = 010$
 $kill_{V,A} = kill_{U,A} = 010$; $kill_{V,B} = kill_{U,B} = 101$

Y.N. Srikant Machine Independent CS & Engg.

Then, we build the region V from U, rather this is a small mistake; from region U by T 1 transformation. The header of U is A and predecessors of U in C and D. So, these are the

two. Now, we need to compute the G set using the two gens. Finally, compute gen of V comma A, V comma B and so on and so forth.

(Refer Slide Time: 56:53)

Results from Iterative RD DFA for the same example

	gen	kill	OUT ₁	IN ₁	OUT ₂	IN ₂	OUT ₃	IN ₃	OUT ₄	IN ₄	RA
A	100	010	100	001	101	101	101	101	101	101	101
B	010	101	010	100	011	101	010	101	010	101	010
C	000	010	000	111	101	111	101	111	101	111	101
D	001	000	001	000	001	101	101	101	101	101	101

$$OUT[B] = gen[B] \cup (IN[B] - kill[B])$$

$$IN[B] = \bigcup_{P, \text{predecessor of } B} OUT[P]$$

$$IN[B] = \emptyset \text{ (initialization)}$$

Reaching Definitions Problem

Y.N. Srikant Machine-Independent Optimizations

Finally, to compare the iterative approach and this region based approach, I have listed here the out and in sets for the iterative data-flow analysis. In the fourth iteration, there is convergence and we produce exactly the same information as in the iterative data-flow analysis. These two are actually identical; it is just that they are two different approaches for the same problem.

(Refer Slide Time: 57:29)

Region Building by T2 Trans. - Available Exp

Basic regions
 $gen_{R,B} = gen[B]$
 $kill_{R,B} = kill[B]$

Region building by T2

For basic blocks B within R1,
 $gen_{R1,B} = gen_{R1,B}$
 $kill_{R1,B} = kill_{R1,B}$

Edges from R2 to header of R1 are not part of R

For basic blocks B within R2,
 $gen_{R2,B} = gen_{R2,B} \cup (G - kill_{R2,B})$
 $kill_{R2,B} = kill_{R2,B} \cup (K - gen_{R2,B})$
 where, $G = \bigcap gen_{R1,P}$, and
 $K = \bigcup kill_{R1,P}$
 for all predecessors P of the header of R2 in R1

For available expressions problem

Y.N. Srikant Machine-Independent Optimizations

The available expression analysis problem is very similar. It is just that in this particular case, the gen and kill definitions become G equal to intersection and K equal to union because available expression information has this.

We will stop at this point and continue this example in the next lecture. Thank you.