Compiler Design

Prof. Y. N. Srikant

Department of Computer Science and Automation

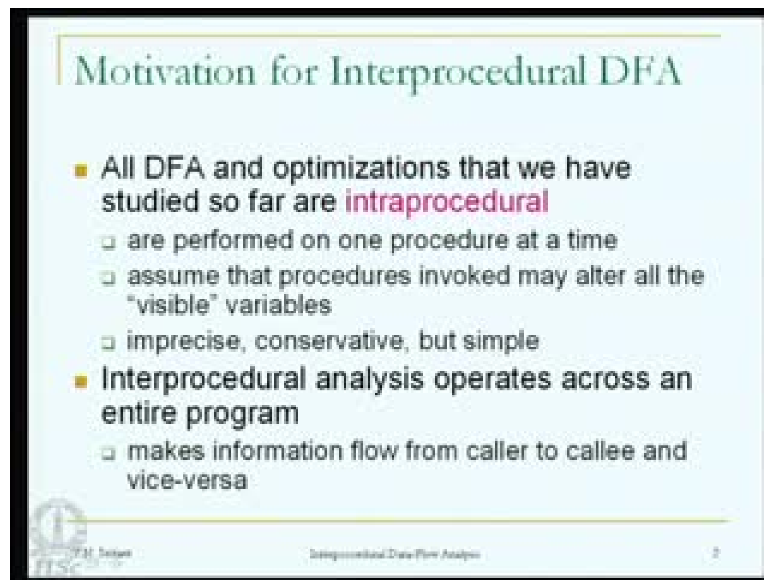
Indian Institute of Science, Bangalore

Module No. # 20

Lecture No. # 38

Interprocedural Data-Flow analysis

(Refer Slide Time: 00:20)



Welcome to the lecture on interprocedural data flow analysis. First of all we need to understand why interprocedural data flow analysis is needed. So, far all the data flow analysis and optimizations that we have studied were all intraprocedural in nature. The point is all these analyses were performed on one procedure at a time. We did not consider a mix of procedures at any point in time. We always took one procedure analyzed it completely and then went to the other and so on.

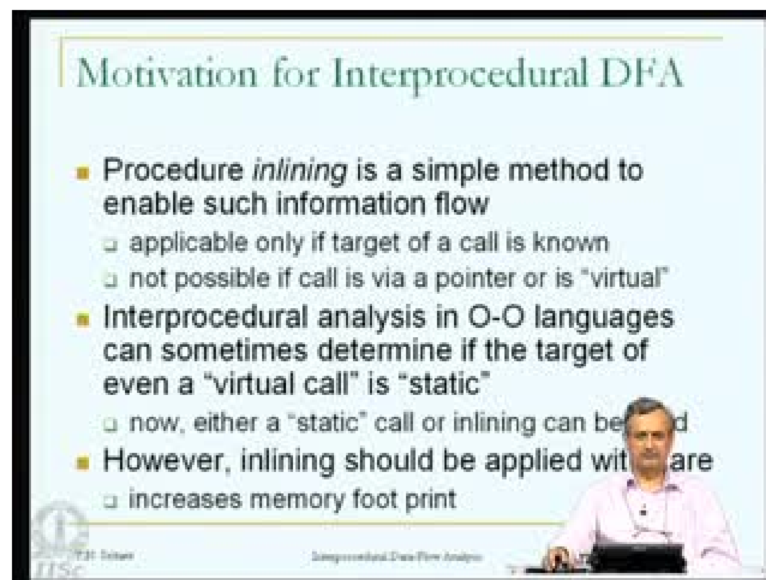
So, the interaction between the procedures when procedure calls happen was not taken into account. So, for example, one of the procedures passes a parameter which is modified by the other etcetera were not taken into account. We assumed that procedures invoked may alter all the visible variables. This was essential because once we ignore the interaction between procedures and consider procedures independently. The side effect of each procedure call must be assumed to be the worst. So, the worst is that it may alter

all the variables visible. So, except for the local variables of the procedure all others were assumed to be modified.

So, such intraprocedural analysis is in general somewhat imprecise. So, it is conservative, but it is very simple; in other words, it happens very quickly. The time required for the analysis is not much, but at the same time the effect of the other procedures etcetera is not incorporated. So, results are imprecise and conservative.

Interprocedural analysis operates across an entire program. It considers the effects of all the procedures. So, it makes information flow from caller to callee and vice versa. So, parameters flow from the caller to callee and then the results flow back or the reference parameters which are modified flow back from the callee to the caller.

(Refer Slide Time: 02:57)



Procedure inlining is a very simple method to enable such information flow. So, what is inlining? We have already studied this. So, basically, in a when you consider a procedure call the procedure corresponding to that particular call you know the body of that could be made to replace the call itself. So, what is involved here? The formal parameters now would be assigned the values of the actual parameters through assignment statements and then the body of the procedure follows. Finally, the result is also conveyed through similar assignment statements.

So, the call is replaced by the body. So, that means, there is actually increase in code size. One has to be very careful about such code size increases. So, this inlining is applicable only if the target of a call is known. So, in other words, if we say a call test with some parameters then, it is possible to inline the procedure test at that point. Suppose the same procedure test is actually called via a pointer; so, we are calling the procedure via a pointer.

We have no idea what the pointer may point to? So, in such a case inlining is not possible. Otherwise, if it is a virtual call for example, in object oriented languages there are methods which are called virtually; we have no idea whether the virtual call points to any one of the possibilities at compile time. So, in such cases inlining is not possible. Interprocedural analysis in object oriented languages can sometimes determine if the target of even a virtual call is actually a static call. In other words, the call may be actually virtual, but the analysis may say that it cannot point to anything, but one particular method.

So, in such a case it is a static call. We can replace the virtual call by a static call. But why should we do this? We know that the virtual calls happen using the dispatch tables so it is there is some indirection and therefore, a cost attached to it. So, we can make it cheaper by making it a static call. So, either a static call or inlining can be used at this point. However, inlining should be applied with care. I already mentioned this because it increases the memory foot print.

(Refer Slide Time: 05:52)

Applications of Interprocedural Analysis

- Convering virtual method calls to static method calls
- Interprocedural pointer analysis helps in making "points-to" sets more precise
  - reaching definitions, available expressions etc., can now be computed with more precision
- Interprocedural analysis eliminates spurious data dependencies, interprocedural constant propagation makes loop bounds known
  - exposes more parallelism during parallelization
- Interprocedural analysis helps in detecting
  - lock-unlock pattern of critical regions
  - disable-enable of interrupts
  - SQL injection (lack of input validation in Web applicatio
  - vulnerabilities due to buffer overflows (frequently, array ds are not checked)

What are the applications of interprocedural analysis? So, converting virtual method calls to static method calls is 1 that I already mentioned. Then interprocedural pointer analysis helps in making points to sets more precise so, a pointer may be pointing to many objects a, b, c, d etcetera. In the worst case, we will assume in the largest set possible at that point, but with interprocedural analysis, we may be able to cut down the size of such sets and make the information more precise.

So, reaching definitions available expressions etcetera can now be computed with more precision. So, they are going to be more accurate. So, that means, the effect of these on optimizations will also be felt. Interprocedural analysis eliminates spurious data dependencies interprocedural constant propagation makes loop bounds known so, these are all possibilities.

So, suppose we are doing auto parallelization; the compiler is doing the parallelization. So, then we compute the data dependence graph. So, in the absence of precise information we may actually, construct the graph with many data dependencies which are actually spurious. So, this can be avoided and the data dependence graph can be made more accurate and precise.

Similarly, if there are loops with whose bounds actually are parameters of the procedure in which it is being executed? The bounds may not be known without an interprocedural analysis, but with such analysis if the loop bounds become known and they become

constants then parallelizations of such a loop may be simpler. So, exposes more parallelism during parallelization. So, that is the general idea.

Interprocedural analysis helps in detecting for example, lock unlock pattern of critical regions. So, one may actually do a lock, but may forget to do a unlock on a semaphore. So, such things can be detected by an interprocedural analysis. Disable enable of interrupts so, every disable must be followed by an enable of an interrupt otherwise, interrupts will never happen again. So, this can also be checked by interprocedural analysis.

So, you may wonder why interprocedural? It may also be intra, but in general, there could be procedure calls in between the lock and unlock or disable enable etcetera. So, that is the reason why we require such interprocedural analysis.

Then SQL injection attacks can be detected. So, what is SQL injection attack? For example, in web applications, the input may not be validated properly so, the account number and the password may not be validated properly. So, it is well known that if the account name or the number is some weird set of characters such as percent hash at etcetera.

Some of the mechanisms login mechanisms do not check whether it is a valid name whether it requires a password etcetera. They actually, login such users with some any password or without even a password. So, then the user gets control of the information and can create havoc. So, this is through SQL injection and software vulnerabilities due to buffer overflows for example, so in general, c plus programs may not check the array bounds at run time and definitely not during compile time. So, if there is an overflow and there is some area of the buffer, which is freely available for writing some information the user may be able to write some extra information by adding something and then he may he or she may be able to access sensitive areas.

(Refer Slide Time: 10:17)

Call Graphs

- A call graph for a program is a set of nodes and edges such that
  - There is one node for each procedure
  - There is one node for each call site
  - If call site c may call procedure p, then there is an edge c → p
- C and Fortran make procedure calls directly by name
  - hence call target of each invocation can be determined statically

So, those were that is about the application of interprocedural analysis. So, let us talk looking at the analysis itself. We need to understand what exactly is a call graph. A call graph for a program is a set of nodes and edges as usual because it is a graph. So, there is 1 node for each procedure. There is 1 node for each call site; then if call site c may call procedure p then there is an edge c to p. So, I will show you an example very soon.

(Refer Slide Time: 11:08)

Call Graphs

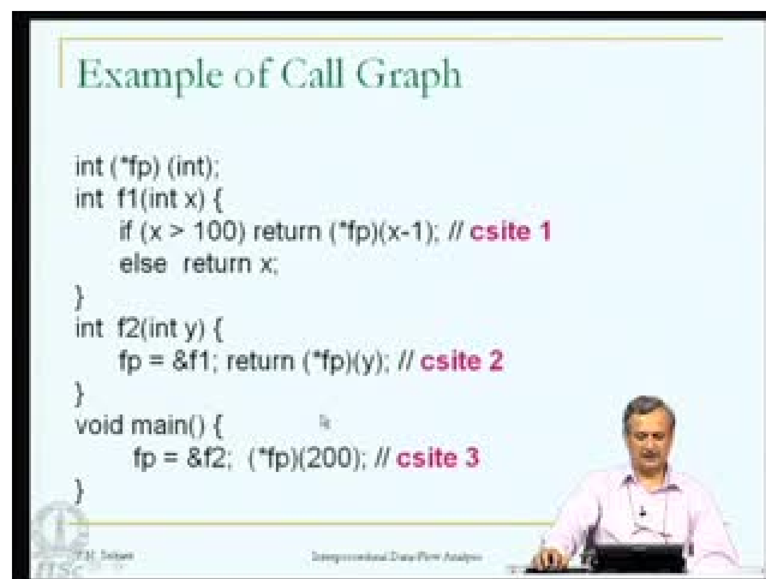
- If the program includes a procedure parameter or a function pointer
  - target is not known until runtime
  - target may vary from one invocation to another
  - call site can link to many or even all procedures in the call graph (considering only return types of functions)
- Ex: virtual method invocations in C++/J
  - calls through the base class pointer canno resolved till runtime

C and Fortran make procedure calls directly by name. So, hence, call target to such invocation can be determined statically otherwise, as i already mentioned, you cannot do

that. If the program includes a procedure parameter or a function pointer then the target may not be known until run time. So, target may vary from 1 invocation to another. The call site can link to many or even all procedures in the call graph. So, considering only return types of functions we have no other information. So, we should probably force to attach to anything.

So, this is, since we are calling through a pointer; a pointer may point to anything, that is nothing better you can do. We will have to point to a very procedure that is available in the graph. So, for example, virtual method invocations in c plus plus and java are of this kind. Calls through the base class pointer cannot be resolved till runtime.

(Refer Slide Time: 12:01)



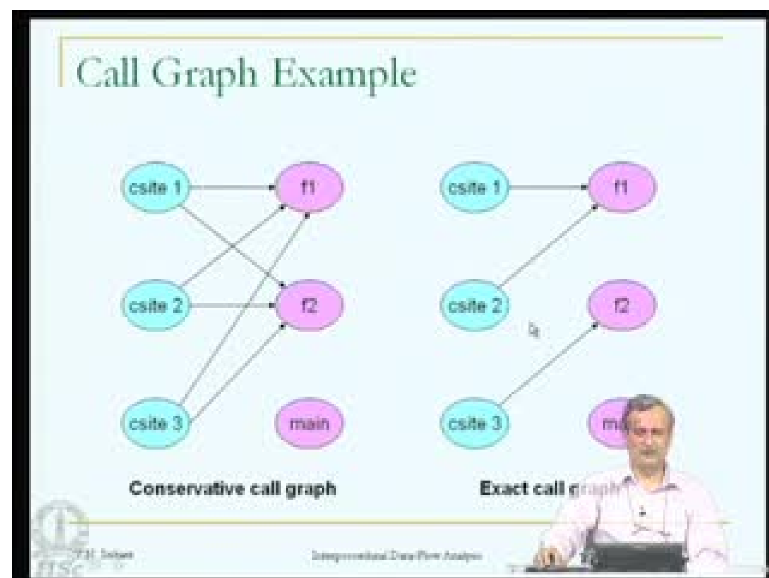
So, let us take a simple example. There is a function called f 1; f p is a function pointer. So, f 1 takes an integer parameter. It checks, if x greater than zero; then return star f p x minus 1 or return x. So, here, this is a call through the function pointer. And f 2 is similar; f p is assigned f 1; so, f p now points to the function f 1; then you return star f p y. So, again, it is a call through the function pointer.

So, main assigns f 2 to f p and calls through f p. Now, let us understand how the program works. The first time f p is pointing to f 2; so, we have it is really a call to f 2 with a parameter 200. So, we come here; here f p is modified to point to f 1; then it calls through f p again now we are calling f 1. So, when we go to f 1; we check whether, x

greater than 100; now f p is pointing to f 1; that is, a recursion f p of x minus 1 else return x.

So, this is the call sequence. Now in the absence of so, at this point in time we know that this is a call to f 2 and nothing else; at this point in time we know that this is a call to f one and nothing else; and here we definitely know that it is nothing, but a call to f 1. So, that is, if we know the information that what f p points to we can actually, construct the graph very accurately otherwise, it becomes conservative as we see now.

(Refer Slide Time: 14:05)



So, here is the call site 1, call site 2, call site 3, f 1, f 2 and main. So, in the absence of any pointer interprocedural information we do not know what the function pointer points to. So, for each of the call sites, it is actually pointing to both f 1 and f 2. We cannot point to main anyway, because main is a main program form which the call the program starts and no calls can be made to main.

So, f 1 and f 2 are both pointed to by call site 1 similarly, call site 2 also points to both f 1 and f 2 similarly, call site 3 also points to both f 1 and f 2. This in the absence of any information that f p points to a certain function. But once we know that call site 2 always points to f 2, call site 2 always points to f 1 and call site 1 always points to f, we can define this particular call graph to be this. This is an exact call graph. So, this is an example of call graph.

(Refer Slide Time: 15:13)

Analysis of Call Graph

- Presence of references or pointers to functions or methods
  - helps us in getting a static approximation of the values of all procedure parameters, function pointers, and receiver object types
- With interprocedural analysis
  - more targets can be discovered and new edges can be inserted into the call graph
- This iterative procedure is repeated until convergence is reached

So, what do we do with the call graph? Presence of references or pointers to function or methods, it helps us in getting a static approximation of the values of all procedure parameters function pointers and receiver object types. So, if we have any assignments then 2 function pointers as we saw in the example. That is the starting point of a static approximation. So, we start from that point we do not have any other edges at a already added to the call graph. We add only those edges which are possible by a static approximation.

Then with interprocedural analysis more targets can be discovered and new edges can be inserted into the call graph. This iterative procedure is repeated until convergence is reached. So, basically, we need to discover how the calls proceed. So, in other words, we need to trace the call sequence and then construct the call graph accurately.

(Refer Slide Time: 16:23)

There are many aspects of this analysis and one of them is context sensitivity. So, let us understand what is sensitivity. Here is a small program, i equal to 9; while i greater than or equal to 0; t 1 calls test with 100; t 1 is assigned the value of return of this function with parameter hundred; second statement involves test call with 200; and the third one with parameter 300.

We have val i minus minus equal to t 1 plus t 2 plus t 3; so, the loop continues until i greater than or equal to zero. Test itself just multiplies v by 2 and returns. So, if we actually, do a analysis carefully, let see how we do a careless or careful analysis. Function test is invoked with a constant in each of the call sites that is very clear 100, 200, and 300. But the value of the constant is context dependent. In other words, if it is perform call site 1, then the value is 100 otherwise, from 2 it is 200 and from 3 it is 300.

So, it is not possible to infer that t 1, t 2 and t 3 are each assign constant values hence, for val i as well, unless you recognize the context that is the call site from which it is the call is made. If it is a very naïve analysis, it would infer that the test can return 200, 400 or 600 from any of the 3 calls. So, in other words, we have no information about the call site. So, we are here in test we do not know whether the call comes from 1, 2 or 3. We know that there can be a call from any one of these sites, but we do not know which 1. In such a case, the possible values of v are 100, 200 or 300 because which call site is involved.

(Refer Slide Time: 19:07)

Context Insensitive Analysis

- Treat each call and return as goto operations
- Create a super control flow graph
  - contains all the normal intraprocedural control-flow edges
  - edge connecting each call site to the beginning of the pocedure it calls
  - edge connecting return statement back to the call site
  - assignment statements to assign
    - each actual parameter to its corresponding formal parameter
    - the returned value to the receiving variable
- Apply standard analysis on the super CFG
- Simple, but imprecise, because a function is analyzed as a common entity for all its calls and only its input-output behaviour abstracted out

So, the values returned by test will be 200, 400 and 600 any one of these or 600. Whereas, if the call site information is available to us it is a context sensitive analysis. It returns 200, 400 and 600 for t 1 t 2 and t 3 and 1200 for val i. So, if we know the call sites then we know exactly, which value is returned for which call site? So, this is called context sensitivity, because the value returned is sensitive to the call site - call site is the context. So, let us understand a little more what context insensitive analysis is.

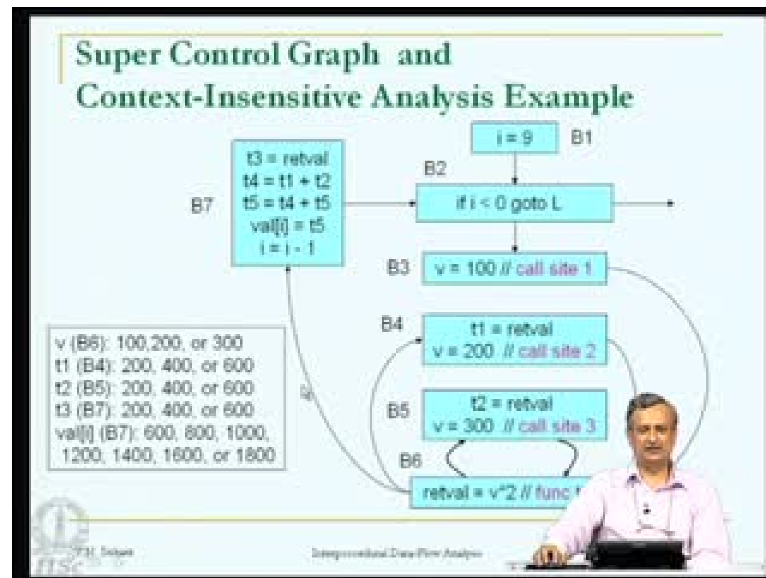
We treat each call and each return as goto operations. It is as simple as that and we create a super control flow graph. So, in other words, we now get rid of procedure call we introduce a goto operation for both call and data. So, what does the super control flow graph contain? It contains all the normal intraprocedural control flow edges. The edge connecting each call site to the beginning of the procedure it calls.

This is a goto operation. Now edge connecting return statement back to the call site, this is another goto operation. And assignment statements to assign the actual parameter to its corresponding formal parameter the returned value to the receiving variable. So, these are all the extra assignment that we need to introduce.

So, now the super control flow graph. I will show you an example very soon; is a simple flow graph, there is no procedure call inside so, we can apply standard analysis - dataflow analysis - on the super control flow graph. So, such a method is very simple, but it is very imprecise; because a function is now analyzed as a common entity for all its

calls and only its input output behavior is abstracted out. We do not distinguish between call sites. Any call site can make an entry into it. So, we are going to be very conservative.

(Refer Slide Time: 20:48)



So, the same previous example, we started off, remember the i equal to 9 and the loop, and test of 100, 200 and 300. So, we start with i equal to 9, then this is the loop control. If i less than 0, goto L, quit the loop; otherwise, we continue with the loop.

So, now the first call to test. The call to test itself is here, it does nothing more than v star 2, return value is v star 2 that is the body of the function test. So, we have v equal to 100; that is the call site 1. We have assigned a parameter, then we make a jump to the body of the function test. Let us say- 100, it computes 200. Now, possibly- we would make a jump to this. So, t 1 equal to 200; this is a call site 2. Then we jump here, we compute to 400 and then jump to this point possibly.

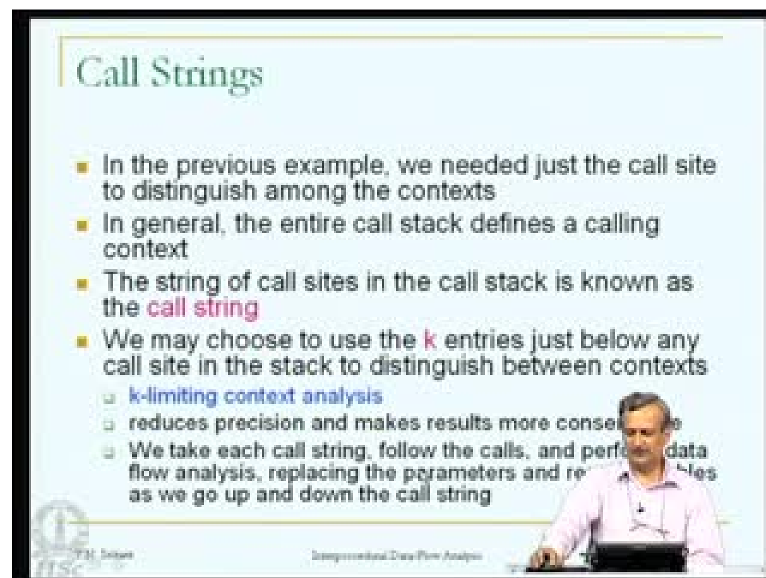
Then again t 2 equal to retval, v equal to 300. We jump here and then we go out. So, you may wonder, why we are going to the appropriate place, but when there are many possibilities from here, we can go to this point (Refer Slide Time: 20:48) or this point or this point. Any one of this is possible. Here, of course, t 3 equal to retval, t 4 equal to t 1 plus t 2 and t 5 equal to t 4 plus t 5. So, this is t 3 sorry t 4 plus t 3 in val i equal to t 5. So, i equal to i minus 1 and then you go back to this when the loop continues.

But now, the value of v at b 6: it could be 100, it could be 200 or it could be 300. We do not know what the value is because you could come from three places. t 1 at b 4 here, for example, if we have come here and then gone there it would have been 200. If you had actually, come to this from here to this point and then gone back, it would have been 400; otherwise, it would be 600.

The same is true for t 2 and t 3 as well. Each of those could be 300, 400 or 600. So, that means, the value of val i at this point is a combination of any of these. So, we have many possibilities for val i: 600, 800, 1000, 1200, 1400, 1600 or 1800.

So, this is all that you can infer from the context insensitive analysis. Because we have no idea that this call site- a particular call site- gives this particular parameter value and gets that particular return value. All that is kind of arrays. Now, you have a super control flow graph. This is very conservative and therefore, imprecise. So, instead of saying val i gets the value of t 1, t 2, t 3 as 200,400, 600 and at the sum is the only value. We have now said: given so many possibilities of 600, 800, 1000 etcetera.

(Refer Slide Time: 24:35)

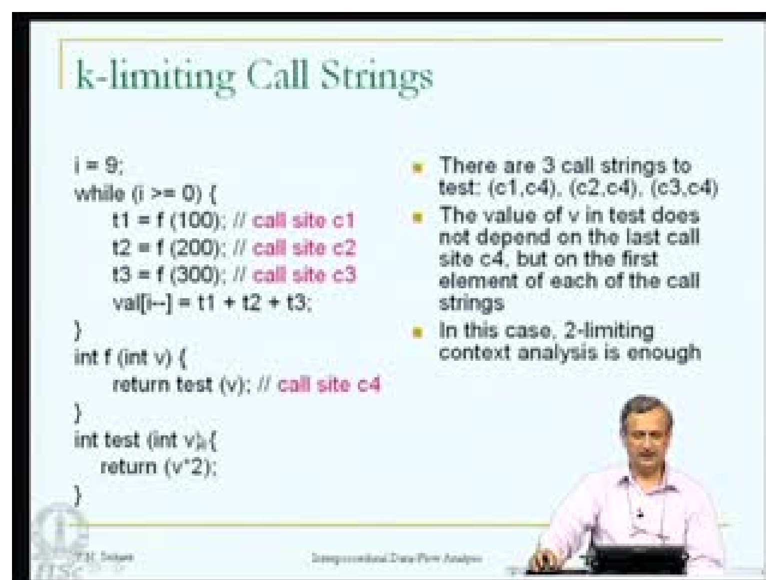


So, val i would have exactly, 1 value if it is a context sensitive analysis, but now it gets 4 plus 4 8 values rather than 3 plus 4 7 values. So, that is the imprecise nature of this context are- insensitive analysis. So, what exactly are call strings? We saw one aspect that is the context sensitivity. So, let us see what call strings are.

So, in the previous example, we needed just the call site to distinguish among the contexts. In general, the entire call stack defines a calling context. Now, we are going to see why the complete stack may be necessary to distinguish context. The string of call sites in the call stack is called as a call string. So, we may choose to use only k entries just below any call site in the stack to distinguish between contexts so, in which case it is called as a k limiting context analysis. So, we are not using the entire call stack, but at any point in time, just the k entries below a particular value in the stack. Those are the k precedent calls calles rather callers. So, those are the only once we are going to consider that becomes a k limiting context analysis.

So, such a k limiting context analysis reduces precision and therefore, makes results even more conservative. So, we take each such call string, follow the calls, start from let us say, main there is a call string, attached to any particular procedure to reach may be many calls strings are possible for a particular procedure. So, we consider all such possible call strings, do an analysis of all the call strings and then determine the various effects of the procedure.

(Refer Slide Time: 26:35)



So, we perform data flow analysis, replacing with parameters, result variables and as we go up and down the call string. So, such an accurate for each call string we are going to make such an analysis and therefore, it is very accurate. Let us see, what k limiting call strings are. So, the same program that we have here. Now, there are 3 call strings only

thing is there is a small modification. Instead of test, we have another function f which is called here and f intern calls test. So, we always call f and f calls attach. There are 3 call strings now to test. If you want to each test you have to first call f and then call test so, it could be from c 1 and then c 4 or c 2 and c 4, c 3 and c 4. So, these are the 3 call strings possible for the procedure test. Value of v in test does not depend on the last call c 4 so, this is a last call. In the previous case, we did not have f so, it dependent on just the last call. Now, there is c 4 as well so, but it depends on 1 call before that is c 3 or c 2 or c 1. So, in this case, 2 limiting context analysis here now, but 1 limiting context analysis is not enough to determine the accurate result.

(Refer Slide Time: 27:51)

Complete Call Strings

```
i = 9;
while (i >= 0) {
    t1 = f (100); // call site c1
    t2 = f (200); // call site c2
    t3 = f (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f (int v) {
  if (v > 101)
    return f (v-1); // call site c4
  else
    return test (v); // call site c5
}
int test (int v) {
  return (v*2);
}
```

- There are 3 call strings to test
- (c1,c5), value returned is 200
- (c2,c4,c4,....,c4,c5): c4 is repeated 100 times, value returned is 202
- (c3,c4,c4,....,c4,c5): c4 is repeated 200 times, value returned is 202
- The value of v in test depends on the full call string
- In this case, k-limiting context analysis is not enough for any k

So, we may require actually, more than 2. In general, so, let us see a demo of that. So, f has been modified this part remains the same. So, f says, if v greater than 101 call f recursively f of v minus 1 otherwise, call test. Test remains as it is. Now, again let us look at test, so, we start with f and it is 100. So, v actually, is less than 101 so, it just returns calls test. So, that means c 1 and c 5. So, this is the call string for this particular one of this c 1 and c 5 this is a 1 call string value returned is 200.

Then suppose, we start with c 2 in that case, it is 200. So, we would have actually, be called f 100 times, then the value reduces to 101 and finally, we call test. So, c 2, c 4, c 4, c 4, c 4, c 4 repeated 100 times is the call string, value returned is 202.

Similarly, for c 3, we would have c 3, c 4, c 4, c 4 etcetera c 4 is repeated 200 times and then, we call it as so. The value returned is again 202 because the value come becomes 101. So, the value of v in test depends on the full call string not either 1 or 2 or 3 limited call strings so, because if the instead of this being 100, 200, 300. Suppose, this was some other value then, as many times this recursion would have been repeated as many times so, the costing would have become much longer.

(Refer Slide Time: 30:00)

Cloning-Based Context-Sensitive Analysis

Simple, context-insensitive analysis is enough on the cloned call graph

```
i = 9;
while (i >= 0) {
    t1 = f1 (100); // call site c1
    t2 = f2 (200); // call site c2
    t3 = f3 (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f1 (int v) {
    return test1 (v); // call site c4.1
}
int test1 (int v) {
   return (v*2);
}
```

```
int f2 (int v) {
    return test2 (v); // call site c4.2
}
int test2 (int v) {
   return (v*2);
}
int f3 (int v) {
    return test3 (v); // call site c4.3
}
int test3 (int v) {
   return (v*2);
}
```

Recursive programs cannot be handled

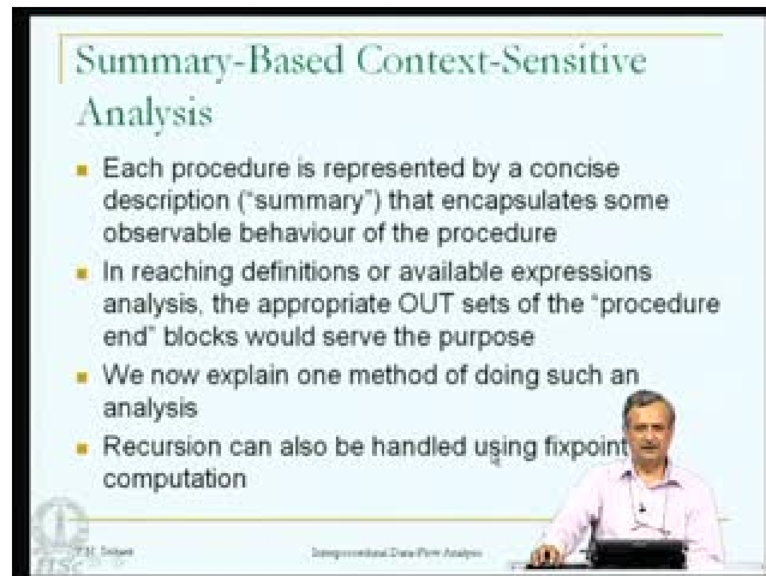
So, this is an example, to show that k limiting costing for any k is in general not sufficient to analyze a program. It may require the full calls site the entire cost string. Now, suppose, we want to do context sensitive analysis, we know that context insensitive analysis is not very useful. So, let us understand, how to conduct context sensitive analysis.

First approach is what is known as Cloning-Based Context-Sensitive Analysis. So, this is very simple. What is cloning? We have this program. So, we have f 1 for each one of these call sites, instead of a single function f calling it we are going to introduce a specialized function. So, t 1 has its own function f 1; f 1 calls test 1; so, this is 1 set. Similarly, t 2 has f 2 and f 2 calls test 2. So, this is another set. T 3 has f 3 and f 3 calls test 3.

So, there are 3 different functions f 1, f 2, f 3 and within that 3 different function test 1, test 2 and test 3 are called. So, there is code explosion here. Instead of just 1 function f

and 1 function test, now, we have 3 f functions and 3 test functions. So, there is code explosion. Once we have such cloning, we know that f 1 always calls test 1. So, there is no other ambiguity here. Simple context insensitive analysis is enough on the cloned call graph, but recursive programs cannot be handled by the scheme. So, there is code explosion, but there is accuracy. But recursion cannot be handled properly.

(Refer Slide Time: 31:43)



Then the second approach is called as the Summary-Based Context-Sensitive approach. So, here each procedure is represented by a concise description or summary that encapsulates some observable behavior of the procedure. So, for example, in reaching definitions or available expression analysis, the appropriate OUT sets of the procedure end blocks would serve the purpose of this observable behavior. So, outsets are available to the outside world. So, this is what we want as the observable behavior.

(Refer Slide Time: 33:00)

So, for each of the procedures using some assumptions, we compute the outsets and then they can be appropriately used in order to compute interprocedural information. Then for each call, we are going to actually use this summary information and derive some result. So, let us now look at one such method in some detail. Here, recursion can also be handled, because fix point iteration is possible in this approach. To understand this properly, we need to understand the problem of what exactly are aliases.

So, look at this picture. We have a equal to b plus x here; then we have i equal to c here, and then d equal to b plus x. So, control flows through these basic blocks. This point suppose, this is v plus x; so, if y is not related to any of these either b or x, in such a case, this b plus x and this b plus x are identical there is no problem. But if this y is an aliase of either b or x-that is the name y stands for either b or x, it is possible to have such aliases. We will know very soon how aliases arise.

So, just assume that name y is an aliase of either b or x. If y is modified, b is modified or if y is an aliase of x then if y is modified x is modified. So, because we have y equal to c here that means it is a modification of the expression b plus x. So, this b plus x and this b plus x are not the same. If such is the case, then we cannot say that b plus x is an available expression at this point. So, that is why aliase analysis becomes important.

How can aliases arise? Let us consider a procedure. So, let us a procedure p (x,y) and then there is a call to p such as p (z,z). So, the same parameter z is passed twice in for

both x and y or it could be a call of p (u,v) from another procedure q (u,v) but q is called as q (z, z). So, 1 level of indirection q is called as z z z z z then, but p would be called as with in p (u,v) itself. But since v and u are both z it is the same as this in effect.

(Refer Slide Time: 35:24)

Aliases

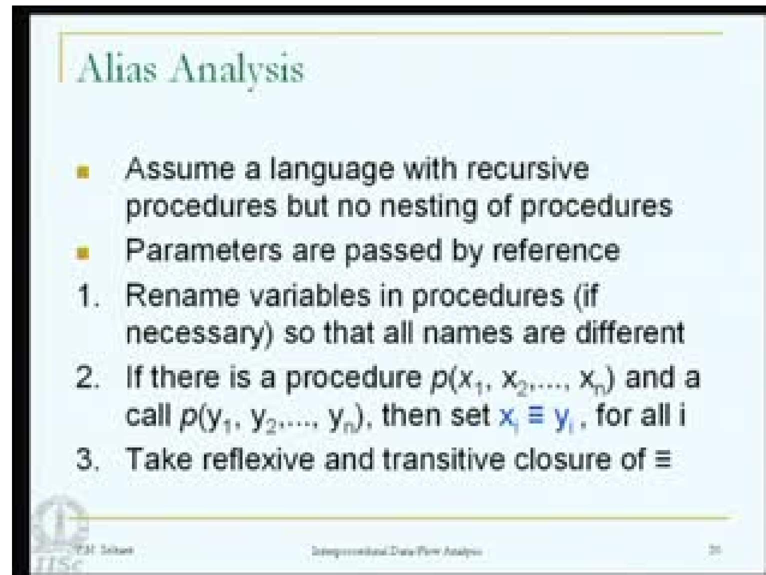
- In reaching definitions, it is conservative not to regard variables as aliases when in doubt
  - So, we do not kill definitions when in doubt
- But, in available expressions, it is exactly the opposite
  - In the above example, if b+x is to be available in B3, we must be *certain* that b and x are no aliases of y
  - If in doubt, we assume aliasing and kill

So, that is how aliases really arise. So, say what happens is, we have x and y here 2 parameters, but we have the same actual parameters at here, both x and y stands for z. If x is modified, z is modified, if y is modified again z is modified and the same is true here u and v both stands for z. So, this is how aliases arise in program. In reaching definitions, it is conservative not to regard variables as aliases when in doubt. So, we do not kill definitions when in doubt. So, it is to say one more definition reaches even though it does not. But in available expression, analysis it is exactly the opposite.

So, for example, in the previous program if b plus x is to be available in b 3, we must be certain that both b and x are not aliases of y. So, if they were, then since y is modified then b plus x would have been modified. So, we cannot in available expression analysis it is a must analysis so, it is not enough if v plus x reaches along one path; v plus x must reach along all path. Even if there is a little bit of doubt that b plus x is going to be modified, we are not going to consider it as an available expression. If in doubt, we assume aliasing and we kill b plus x. So, in the point here is, the application the reaching definitions or available expressions etcetera will determine whether aliasing is harmful or

is not harmful. So, we just compute alias information and let the application use it the way it really wants.

(Refer Slide Time: 37:24)



How is alias analysis done? Actually, with names it is a fairly straight forward procedure. Let us assume a language with recursive procedures, but no nesting of procedures. So, if they are local parameters aliases cannot happen. Let us assume that parameters are passed by reference only. Step number one: rename the variables in procedures if necessary so, that all names are different. There are no names which are being shared across procedures. All parameters dummy and actual will have different names.

So, if there is a procedure p with parameters x 1 to x n; a call to p with parameters actual parameters y 1 to y n; so, these are all call by reference. So, there are only variables y 1 to y n. There are no expressions in place of these variables. Now y 1 and x 1 are aliases, y 2 and x 2 are aliases, y n and x n are also aliases. So, we are going to set x i is an alias of y i, this is a symmetric relation. So, if x i is an alias of y i, y i is also an alias of x i.

(Refer Slide Time: 38:56)

Alias Analysis Example

```
global g,h;
  procedure main() {
    local i;
    g = ...; one(h,i);
  }
  procedure one(w,x) {
    x = ...;
    two(w,w); two(g,x);
  }
procedure two(y,z) {
  local k;
  h = ...; one(k,y);
}
```

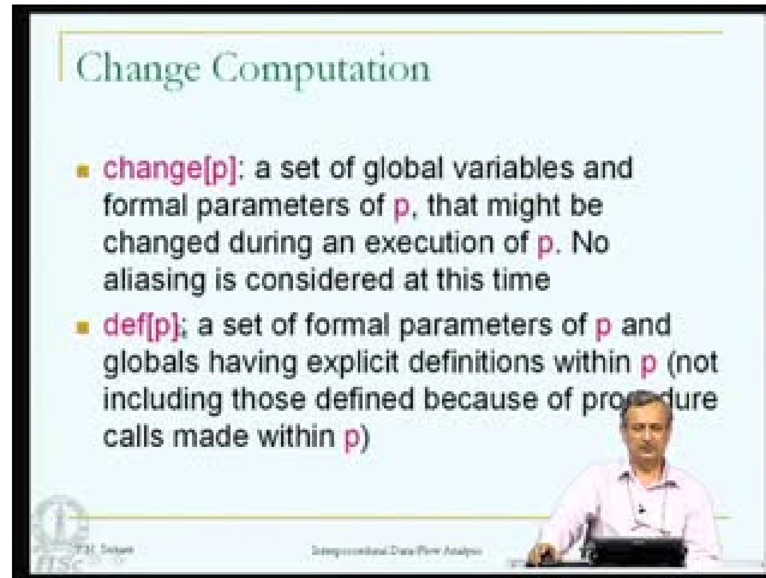
- **main**: h ≡ w, i ≡ x
- **one**: w ≡ y, w ≡ z, g ≡ y, x ≡ z
- **two**: k ≡ w, y ≡ x
- All variables are aliases of each other

We said this for all i, this is an equivalence relation really. We take the reflexive and transitive closure of this alias relation and we get the aliases for all the variables. So, let us take an example. So, g and h are global variables. Procedure main as a local variable i; then we have assignment to g we call 1 with h comma i. Then there is procedure one with 2 variables (w,x), x is assigned a value. We call 2 with (w,w); we call 2 with (g,x).

So, procedure 2 has a local variable k; h is assigned a value; then 1 is called with k and y. Let us start doing the alias analysis on main. For alias analysis only procedure calls are important. So, we go to the call 1 (h,i); h and i are local are actual parameters. Let us take h in the procedure 1 it corresponds w; so h is alias of w. Let us take i; i is now alias of x. Similarly, if we take this call w is an alias of y and again w is an alias of z as well. Now, in this case g is an alias of y and x is an alias of y. That is also correct. In this call k is an alias of w and y is an alias of x.

So, now, if you take the transitive closure reflexive transitive closure, we see that all variables are aliases of each other. So, you can trace that for example, h equal to w, w equal to y, h equal to y and w equal to z, h equal to z. So, then y equal to g, h equal to g, y equal to x, h equal to x. So, h is related to all of them.

(Refer Slide Time: 40:47)



Now, one possible observable behavior is the change set of a procedure. What exactly is this change set? So, change of p, say set of global variables and formal parameters of the procedure p that might be changed during an execution of p. We are going to look at only reference parameters here, not local non-call by value parameters are not considered. And again only global variables and formal parameters can transmit effect outside the function or procedure p. So, we are interested only in the set of global variables and formal parameters of the procedure p.

That might be changed during an execution of p. No aliasing is considered at this point. So, this is the set that we want to compute using inter procedure analysis. When we are computing p in this change of p, we need to consider the procedure calls within p as well.

Now, this requires 3 parameters; change of p will require def of p; another set called a of p and the third one called g of p. Let us define them. So, change of p cannot be computed once for all it has to be done through an iterative process. What is def of p? A set of formal parameters of p and globals having explicit definitions within p. So, not including those defined because of procedure calls within p. We do not worry about procedure calls within p. When we want def p we just take the assignments to globals, assignments to formals, within p and that is the def p.

(Refer Slide Time: 42:59)

Change Computation

- change[p] = def[p] ∪ A[p] ∪ G[p], where
- A[p] = {a | a is a global variable or formal param of p, such that, for some proc q and integer i, p calls q with a as the ith actual param and the ith formal param of q is in change[q] }
- G[p] = {g | g is a global in change[q] and p calls q }
- We use a simplified calling graph whose nodes are procedures. There is an edge from p to q if p calls q somewhere in the program

(Refer Slide Time: 43:13)

Example for the set A[p]

```
procedure p(...)
{ call q(...,a,...)
  ...
}
```

i^{th} actual parameter

```
procedure q(b1,b2,...,bi,...,bn)
{  ...
}
```

i^{th} formal parameter and b_i is in chang

This can be computed statically once for all it does not change. Because we are only looking at concrete assignments to the globals and formal parameters. That is what i said change p is def p union a p use in g p. What is a of p? It is actually a simpler to show you a picture and then come back. Let us take a procedure p. So, in its body, it calls a procedure q, so q is here. When we call q, there is a parameter a which is passed and this is the ith actual parameter in this list.

So, now let us go to q. In q, there are n parameters. So, b i is the ith formal parameter, a and b i are the corresponding parameters, a is the ith actual parameter, b i is the ith formal parameter. Now, suppose b i is already in change of q. So, if b i is already in change of q then a will also be in change of q.

The reason is, we are calling q b is definitely going to be changed by b i is going to be changed by this procedure q. Since b i and a correspond to each other, a will also be changed when we call procedure q. So, the effect of procedure q is to change a. So, a goes into the change set of p.

So, a is a global variable or formal parameter of p, such that, for some procedure q and integer i, p calls q with a as the ith actual parameter and the ith formal parameter of q is in change of q. So this is what i was explaining now.

So, we take the procedure call. Take a particular parameter; look at the corresponding formal parameter of that particular call. If that formal parameter is in the change of the called procedure, then put this particular parameter a also into the change set of p. Rather a set of p a goes into change set of p.

(Refer Slide Time: 45:51)

Change Computation

- Input: A calling graph with a collection of procedures, $p_1, p_2, \ldots, p_n$. If the calling graph is acyclic, then we assume that p_i calls p_j only if $j<i$, otherwise, no assumptions
- Output: change[p]
- It is assumed that def[p] is precomputed

What is g of p? So, g is a global in change of q and p calls q. So, p calls procedure q and q. We are now looking at the global variables and q changes this g. So, it is in the change set of q therefore, g will also now be in the change set of p. Because p is calling q if g

changes in q it changes in p as well. We use a simplified calling graph whose nodes are procedures. We do not have call sites as nodes. There is an edge from p to q if p calls q somewhere in the program. So, we do not have any node for the call site. All these are merged. If there is some call form p to q, we just say there is an arc from p to q.

So, the input: is a calling graph with a collection of procedures p 1, p 2, p n. If the calling graph is acyclic, then we assume that p i calls p j only if j less than i, some ordering is possible. Otherwise, no assumptions. Output: is change of p and it is assumed that def of p is precomputed. So, we do not really need this ordering, it can be done as it is also.

(Refer Slide Time: 46:28)

Change Computation

```
for each proc p do change[p] = def[p];
while changes to any change[p] occur do {
  for i = 1 to n do {
    for each proc q called by p_i do {
      1. add any globals in change[q] to change[p_i];// adding G[p]
      2. for each formal parameter x (j^th) of q do
           if x is in change[q] then
             for each call of q by p_i do
               if a, the j^th actual param of the call is a
                 global or formal parameter of p_i then
                 add a to change[p_i] // adding A[p]
    }
  }
```

So, what we really do is start with change of p equal to def p that is the initialization. Then we do a fix point computation while changes to any change p occur do. So, for i equal to 1 to n for each procedure q called by p i, we are considering 1 procedure p i at a time. We look at all the procedures q called by p i. Now, we add g of p i any [cad/add] any globals in change of q to change of p i so, this is adding g of p i. For each formal parameters now we are looking at the computation of a. So, say the jth parameter of q if x is in change of q, then look at the call q by p. If a the jth actual parameter of the call is a global or formal parameter of p i, then add a to change of p i. So, this is adding a of p i.

So, you look at the actual parameter and the corresponding formal parameter. If the formal parameter is in change of change set, add the actual parameter also to the change set. This is adding the a set to change. So, this is a fairly straight forward procedure. Let

us look at the same example and trace it. So, we need to worry about the assignment with g is a global, there is an assignment in main, x is a parameters so there is an assignment in 1, then h is another global with there is an assignment in 2 and there are couple of procedure calls.

(Refer Slide Time: 48:03)

So, here is a calling graph; main calls 1; 1 calls 2 and 2 calls 1, so, its recursive. We start with def main equal to g change this is same as change main. Similarly, g main is 5 and so on and so forth. Suppose, these are the initializations so, change 2 is h, change 1 is x. Let us consider that function 2, 2 calls 1. So, 1 is the call is 1 of (k,y) these are the actual parameters k and y. 1 of (w,x) these are the formal parameters of the call the procedure 1. So, k is a local. We do not worry about the local parameter at all local variable at all. so we are now going to worry about only y and x. So, x is in the change set of 1. So, see here, change set of 1 already contains x. So, y will also be added to the change set of 2.

Now, change set of 2 already had h, now we have added 1 as well that is because of the a c. So, in this manner, we really consider 1 and then add the appropriate parameters there is w w here, y and z here, so, y is already in the change set of 2. So, w is also added to the change set of 2 and so on and so forth.

(Refer Slide Time: 49:57)

Use of Change Information in computing Available Expressions – Method 1

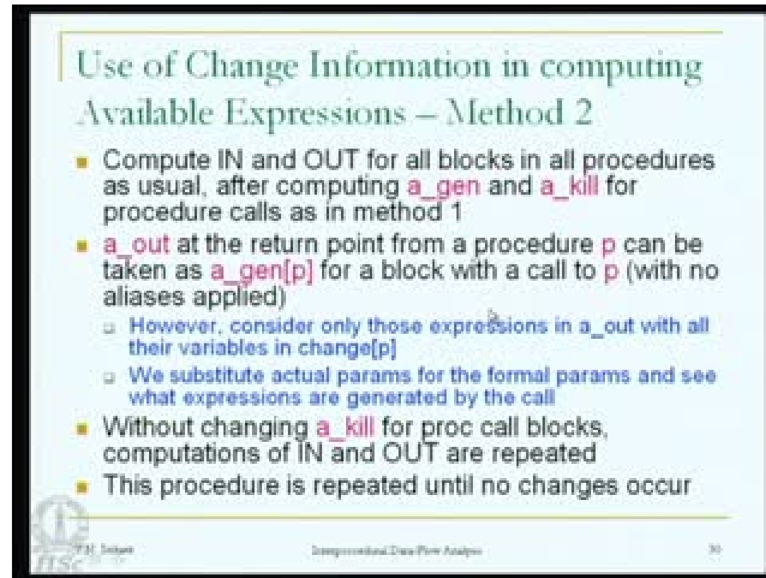
- Each procedure call is a separate basic block
- Method 1: B is a block for call to proc p
  - a_gen[B] = Φ, for all proc call basic blocks
  - a_kill[B]: if a variable b is in change[p], then b kills all expressions involving b and its aliases
  - a_gen and a_kill for all other types of blocks are computed in the usual manner
  - Knowing a_gen[B] and a_kill[B] for proc ca cks, computing IN[B] and OUT[B] for all blocks e whole procedure proceeds in the usual

In this fashion, we propagate the change information from 1 caller to callee and back to the caller and finally, we keep doing it because it is recursion. Keep doing it until there is no further change. So, this is how the change propagation happens. Now, let us see how this change information is used for the available expression analysis- there are two methods by which it can be used.

So, let us assume that each procedure call is in a separate basic block. In method 1: B is a block for the call to procedure p. We assume that a gen of B is phi that means the procedure call does not generate any expressions. So, for all procedure call basic blocks. What about kill? If a variable b is in the change set of p for this procedure corresponding to this block, then b kills all expressions b involving b and its aliases. So, this is where the alias information is becoming useful.

How is this possible? You consider only those variables which are changed by the procedure call by that procedure call. So, only those variables expressions involving those variables will be killed that is very logical. So, a kill of b is computed in this fashion. So, a gen and a kill for other types of blocks are computed as usual, as we studied in the available expression analysis. Now, we know a gen b and a kill b for procedure call blocks and now for all other blocks as well. Computing in b and out b for all blocks in the whole procedure now proceeds in the usual manner.

(Refer Slide Time: 51:50)



So, we had to know now a gen is phi of course, but a kill we could make a better guess rather better estimate. Hopefully, our available expression analysis would be much better because now it incorporates summary information from the procedure calls. So, observe that we are using the change information in computing a kill then we are iterating. This is method 1. In method two: compute in and out for all basic blocks in all procedures as usual, after computing a gen and a kill as in method 1. So, this is the starting point.

After this, see the previous method 1 guess a slightly better estimate for in and out of all the basic blocks. Now we use that as the starting point. How a out at the return point from a procedure p, can be taken as the a gen p for a block with a call to p. So, this is what i was saying. What is that the procedure can generate. Look at its return point or the end point, procedure end point and you can now say that whatever, what was a out at that end or return point is the set of expressions generated for the call block a and it is taken as a gen p for a block with a call to p. No aliases at this point again, but we must consider only those expressions in a out with all their variables in change of p obviously, we should not worry about expressions involving local variables. And we also substitute actual parameters for formal parameters and see what expressions are generated by the call.

So, this is again necessary because we do not want expressions with other variables, which are not in the formal parameter list or some other variables involving locals

etcetera. Actual parameters are the only ones, which matter to us. So, we replace actual parameters, formal parameters by actual parameters and then see these are expressions involving actual parameters. Globals are the only ones which make sense in the procedure after the procedure call is over others do not matter to us. So, that is why this is necessary. So, without changing a kill for procedure call blocks computations of in and out are now repeated.

Now, this procedure is repeated until no changes occur. It is possible that by considering 1 iteration you are feeding to some other procedure. We go on doing this until none of the values change in many of the procedures. So, this is a little more precise and gives you better information than the previous method and of course, both of them give much better information than the one which uses no summary information. So, this is the end of this lecture thank you very much.

.

.