
REPRESENTING CONTROL FLOW
OF PROGRAMS

Control flow

- Control flow describes the rules underlying the evolution of the program
- Control flow is implicit in the program code and the Abstract Syntax Tree (AST) representation:
  - AST represents the *syntactic structure* of the source code in a tree form. Each node corresponds to a construct occurring in the code (e.g., expressions, statements, declarations).

In front end of compiler we build AST. Control flow is implicit in an AST, you don't have mechanisms to represent control flow.

Breaking It Down:

1. **Control Flow Defines How a Program Evolves:**
    - When you write a program, it doesn't just execute everything at once. Instead, it follows specific rules that dictate which part runs next—whether it's moving to the next line, jumping to another function, or looping back to repeat an action.
2. **Control Flow is Implicit in Code & AST:**
    - Your program's structure already *implies* the control flow. For example, if you write an `if` statement, it's clear that some code will execute *only if* the condition is true.
    - Similarly, in an Abstract Syntax Tree (AST), the structure itself represents how the program executes. The AST is a tree representation of the code, where each node is an element of the code (like a loop, function call, or condition).
3. **AST and Control Flow:**
    - The AST is like a *map* of the program's structure. It doesn't explicitly say, "first do this, then that," but its hierarchical structure inherently shows how different parts of the code relate and in what order they should execute.

Control flow Graphs

In the compiler we have a CFG: graphical representation of a control flow of a program - Here you can easily represent the path of execution and splits caused by branching, function calls etc.

- A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications.
  - A CFG is a representation of all possible paths that might be traversed through a program during its execution. Nodes in a CFG are (sequences of) instructions with no jumps and edges represent *control flow* (e.g., jumps, branches, loops, etc.).
- Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside a program unit.
- Control flow graphs were originally developed by Frances E. Allen.

- CFG are used as mechanisms to manage static analysis
  - We set up a set of constraints equations and the idea is to find a fixed point for a function to solve the equations.

Question: can this work (monotone framework we defined) in a more complex program?
Yes with control flow graphs.

Breaking It Down:

1. **What is a Control Flow Graph (CFG)?**
    - A CFG is a *graphical* representation of how a program flows. Instead of looking at raw code or an AST, a CFG shows **all possible execution paths** in a structured, visual way.
    - Each **node** in the graph represents a sequence of instructions that execute one after another, *without jumps or branches.*
    - Each **edge** (or arrow) represents a possible change in control flow, such as a function call, loop, or conditional jump.
2. **Why Use CFGs?**
    - The key advantage of CFGs is that they help in analyzing and optimizing code because they make the structure explicit.
    - They are useful in **static analysis** (analyzing code without running it) and in **compiler optimizations** (like dead code elimination, loop optimizations, etc.).
    - By looking at a CFG, you can see all possible paths the program might take, making it valuable for **security analysis** (e.g., detecting vulnerabilities) and **automated reasoning** about program behavior.

Frances Allen

Frances Elizabeth Allen was an American computer scientist and pioneer in the field of optimizing compilers

Allen was the first woman to become an IBM Fellow, and in 2006 became the first woman to win the Turing Award.

Her achievements include seminal work in compilers, program optimization, and parallelization.

AST vs CFG

Differences: CFG is derived from AST during compilation just before code generation. CFG is more related to intermediate representation of the code

The AST captures the hierarchical structure and syntax of the program, but not its runtime behavior.

The CFG is derived from the AST and reflects the order of execution and possible execution paths.

During compiler analysis, an AST is typically constructed first, and then transformed into a CFG for optimization, static analysis, and code generation.

Breaking It Down:

1. **AST (Abstract Syntax Tree)**
    - Represents the *hierarchical structure* of the code, meaning it captures **what** is written and how different parts relate to each other.
    - Does **not** represent *execution order* or runtime behavior—it's more about **syntax** and **structure** than logic flow.
2. **CFG (Control Flow Graph)**
    - Built *from* the AST and represents the **actual flow of execution**.
    - Shows **possible execution paths**, including loops, conditionals, and function calls, making it more useful for understanding *runtime behavior*.

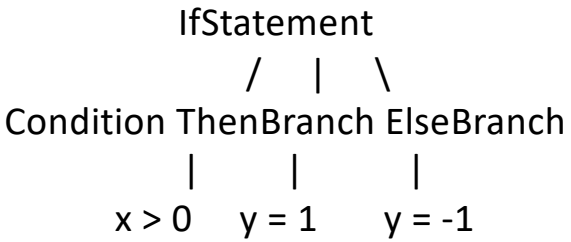
AST vs CFG in Compilation:

- When a compiler analyzes code:
    1. **AST is created first** → This step parses the code into a structured format.
    2. **CFG is then derived from the AST** → This step helps optimize the execution flow, check for errors, and generate machine code.

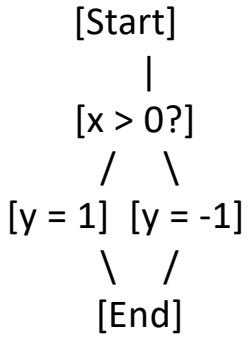
↓

```
if x > 0 {
   y = 1;
} else {
   y = -1;
}
```

PROGRAM



AST



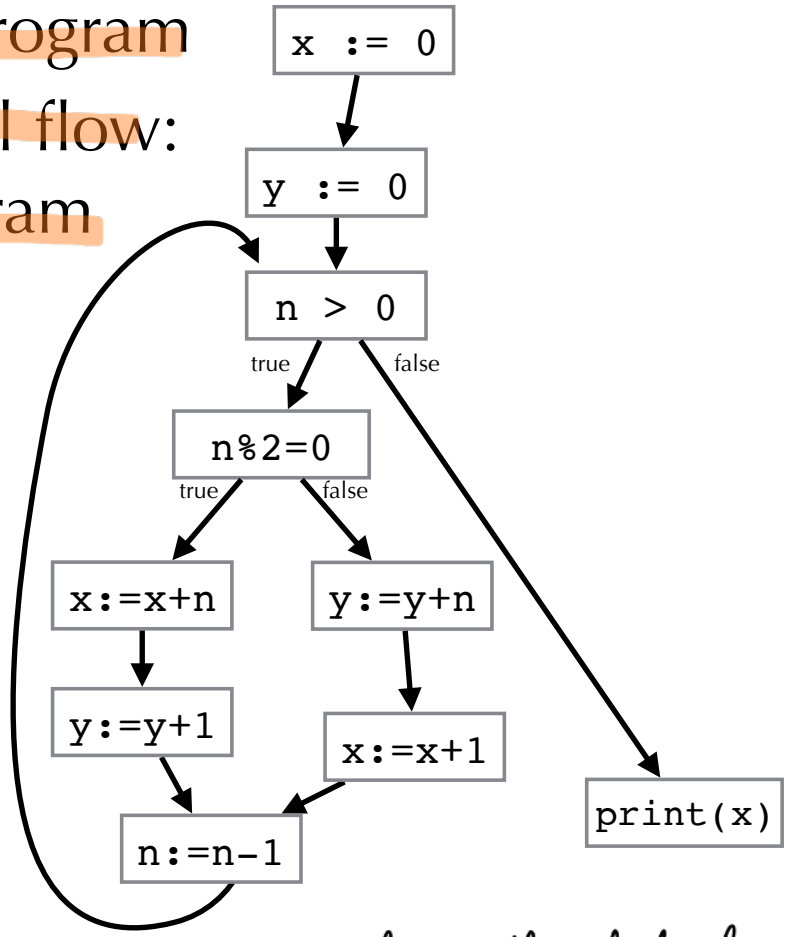
CFG

Graph whose node represent statements.
Arcs represent control flow we can follow.
Then we have a Join, End can be reached by different nodes.

CONTROL FLOW GRAPHS

- Graphical representation of a program
- Edges in graph represent control flow: how execution traverses a program
- Nodes represent statements

```
x := 0;
y := 0;
while (n > 0) {
  if (n % 2 = 0) {
    x := x + n;
    y := y + 1;
  }
  else {
    y := y + n;
    x := x + 1;
 }
  n := n - 1;
}
print(x);
```



while loop with if-then-else inside

Graph concepts manage while / if constructs. VERY INTERESTING

Basic blocks

We don't like number of constructs for each program part. We provide a construct to logical groups that are together.

- **Basic block**: a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning* and leave at the end
  - Sequences of statements
  - Can be entered only at beginning of block
  - Can be exited only at end of block
  - Exit by branching, by unconditional jump to another block, or by returning from function

even for sequential blocks

* May have several incoming arrows, but execution has to start at the beginning of block, and you can exit only at end of sequence of statements.

Breaking It Down:

1. **What is a Basic Block?**
   - A **basic block** is a sequence of instructions that:
     - Always execute **together** (one after another).
     - Have **one entry point** (execution can only start at the first instruction).
     - Have **one exit point** (execution cannot stop or jump in the middle—it must complete the block before moving on).
2. **Key Features:**
   - **No jumps in the middle** → You can't "skip" to a later instruction in a basic block.
   - **One-way execution** → Once a block starts, it runs to the end without interruptions.
   - **Exits via control flow changes** → The block **only** stops when it:
     - Branches (e.g., `if` statements).
     - Jumps to another block (e.g., `goto`, loops).
     - Returns from a function (e.g., `return`).

Example:

Take this simple Python-like pseudocode:

python Copia Modifica

```python
x = 10
y = 5
if x > y:
    z = x - y
else:
    z = y - x
print(z)
```

This breaks into **basic blocks** like this:

vbnet Copia Modifica

```
B1: x = 10
    y = 5
    if x > y → B2 or B3

B2: z = x - y
    goto B4

B3: z = y - x
    goto B4

B4: print(z)
```

- **B1**: Initializes variables and checks the condition (`x > y`). This **branches** into **B2** or **B3**.
- **B2** & **B3**: Each handles an assignment and then **jumps** to **B4**.
- **B4**: Executes `print(z)`, completing the program.

Each **basic block** is an unbroken sequence of instructions, and execution can only enter at the start and exit at the end.

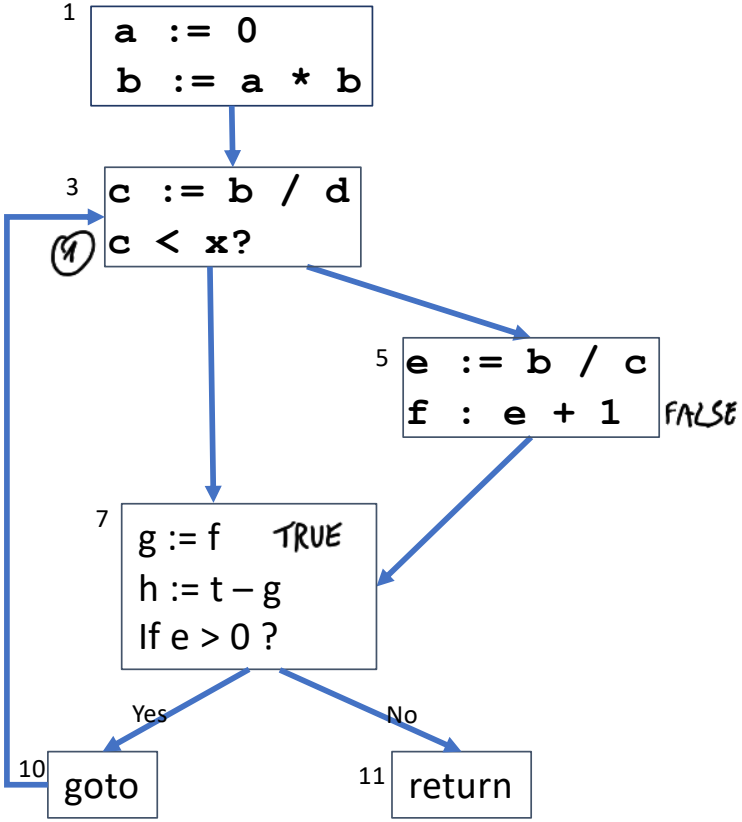
Why are Basic Blocks Useful?

- **Compiler Optimizations**: They help in eliminating redundant computations and improving execution speed.
- **Control Flow Analysis**: Used in **CFGs** to track execution paths.
- **Security & Static Analysis**: Helps in analyzing potential vulnerabilities by tracking execution flow.

Control-Flow Graphs

```
1         a := 0
2         b := a * b
3   L1:   c := b/d
4         if c < x goto L2
5         e := b / c
6         f := e + 1
7   L2:   g := f
8         h := t - g
9         if e > 0 goto L3
10  goto L1
11  L3:   return
```

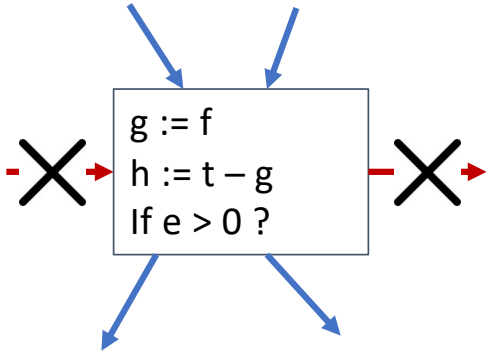
- 6 basic blocks
- ① We have a goto L1, so that has to enter the beginning of a block. If we did 2-3 that was not good.



Basic blocks (operationally)

There is an algorithm that identifies leaders: instructions at the beginning of basic blocks. A block is built starting from a leader and finishing before another leader.

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building basic blocks: Identify **leaders**

- The first instruction in a procedure, or
- The target of any branch, or
- An instruction immediately following a branch (implicit target)

Group all subsequent instructions until the next leader

Breaking It Down:

1. **Basic Block (Reminder)**
   - A **basic block** is a sequence of straight-line code with **only one entry point** (at the beginning) and **one exit point** (at the end).
   - Once execution starts in a basic block, it continues sequentially until it reaches a branch, jump, or return.
2. **How to Identify Basic Blocks?**
   - The process starts by **identifying "leaders"**, which are instructions that mark the beginning of a new basic block.
3. **A leader is:**
   - **The first instruction in a procedure** → Every function or program starts with an entry point.
   - **The target of any branch** → If there's a `goto`, `if`, or loop jump, the destination instruction starts a new block.
   - **An instruction immediately after a branch** → Because execution may continue there after the branch.
4. **Grouping Instructions into Basic Blocks**
   - Once a leader is identified, all following instructions **belong to the same basic block** until another leader is found.

Example

Take this pseudocode:

c Copia Modifica

```
A:   x = 5      // First instruction (leader)
     y = x + 2  // Follows previous, same block
     if y > 10  // Branch (leader)
        goto C  // Ends this block

B:   z = y - 3  // Target of a branch (leader)
     print(z)   // Same block as above
     goto D     // Ends block

C:   z = x * 2  // Target of branch (leader)
     goto D     // Ends block

D:   print("End")  // Instruction after a branch (leader)
```

Identified **leaders** (marked with `// leader`):

go Copia Modifica

```
A:  x = 5       // leader
    y = x + 2
    if y > 10   // leader (branch)

B:  z = y - 3   // leader (target of branch)
    print(z)
    goto D

C:  z = x * 2   // leader (target of branch)
    goto D

D:  print("End")  // leader (after a branch)
```

Each **leader starts a basic block**, and instructions **continue in the block** until another leader is found.

Why is This Important?

- This method allows **breaking down programs into structured execution units**.
- **Compilers and optimizers** rely on this to analyze and transform code efficiently.
- **Control Flow Graphs (CFGs)** are built using these **basic blocks**.

So, in summary: **find leaders → group instructions until the next leader → define basic blocks!** 🚀

Example

```
1         a := 0
2         b := a * b
3   L1: c := b/d
4         if c < x goto L2
5         e := b / c
6         f := e + 1
7   L2: g := f
8         h := t - g
9         if e > 0 goto L3
10  goto L1
11  L3: return
```

Leaders

– {1, 3, 5, 7, 10, 11}

Blocks

– {1, 2}
– {3, 4}
– {5, 6}
– {7, 8, 9}
– {10}
– {11}

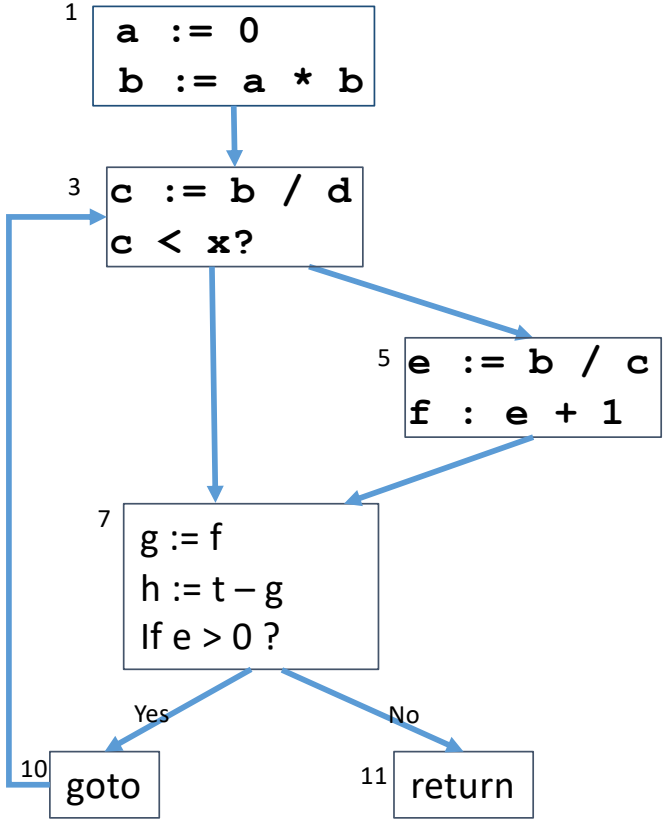
3. can be reached by 10 and statement 2. We have different ways of starting its execution. So it is a leader.

- When from a point we can move to multiple points of program, that is the end of the block.

Building a CFG From Basic Block

Construction

- –Each CFG node represents a basic block
- –There is an edge from node i to j if
  - –Last statement of block i branches to the first statement of j, or
  - –Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)



This slide explains how to construct a **Control Flow Graph (CFG)** from **basic blocks**. The idea is simple: each **basic block becomes a node** in the CFG, and **edges** are added based on how execution flows between them.

Breaking It Down:

1. **Each CFG Node = A Basic Block**
   - Instead of tracking individual instructions, we treat each **basic block** as a **single node** in the graph.
2. **Edges Represent Flow Between Blocks**
   - There is a **directed edge** from **block i** to **block j** if:
     - **Branching occurs** → If the last statement in `block i` is a branch (e.g., `if`, `goto`), it connects to the target `block j`.
     - **Fall-through occurs** → If `block i` does **not** end in an unconditional jump (e.g., `goto` or `return`), and the next block in program order is `block j`, execution naturally "falls through" into it.

Example

Take this simple pseudocode:

c Copia Modifica

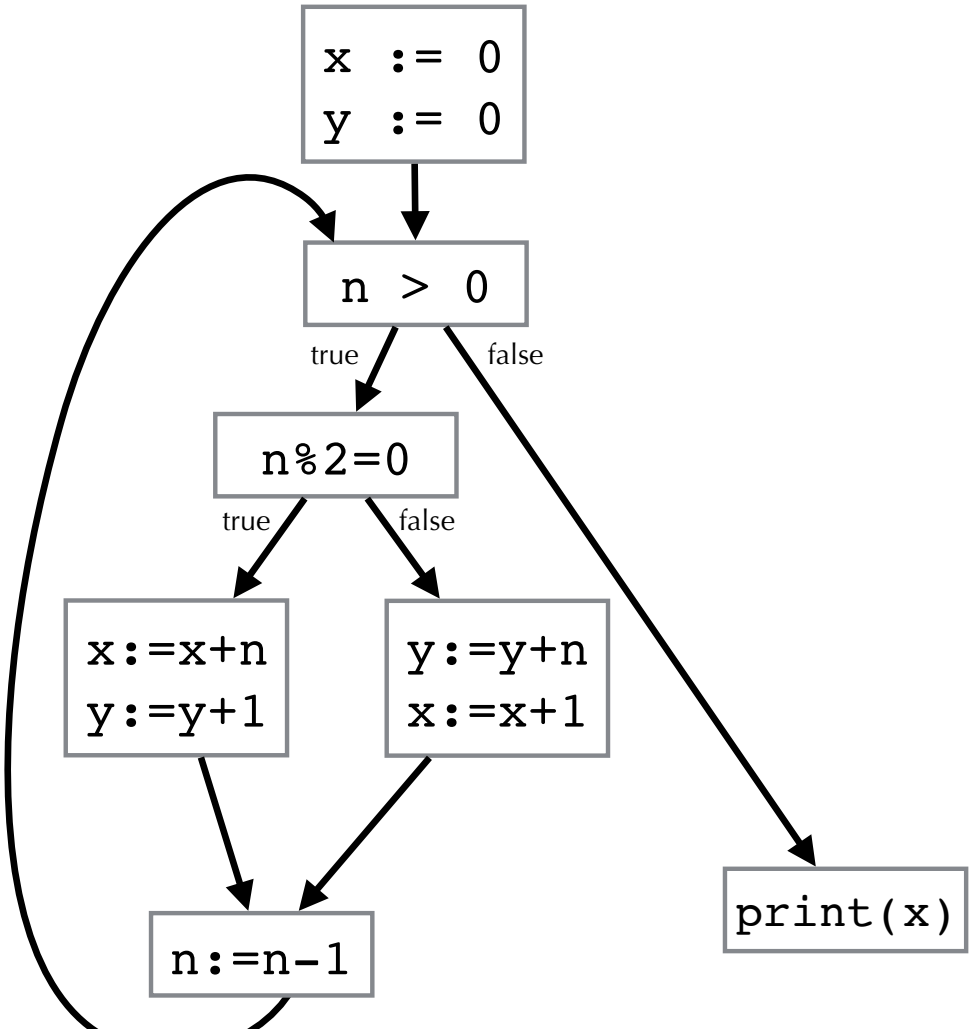
```
A:  x = 5
    y = x + 2
    if y > 10  goto C

B:  z = y - 3
    print(z)
    goto D

C:  z = x * 2
    goto D

D:  print("End")
```

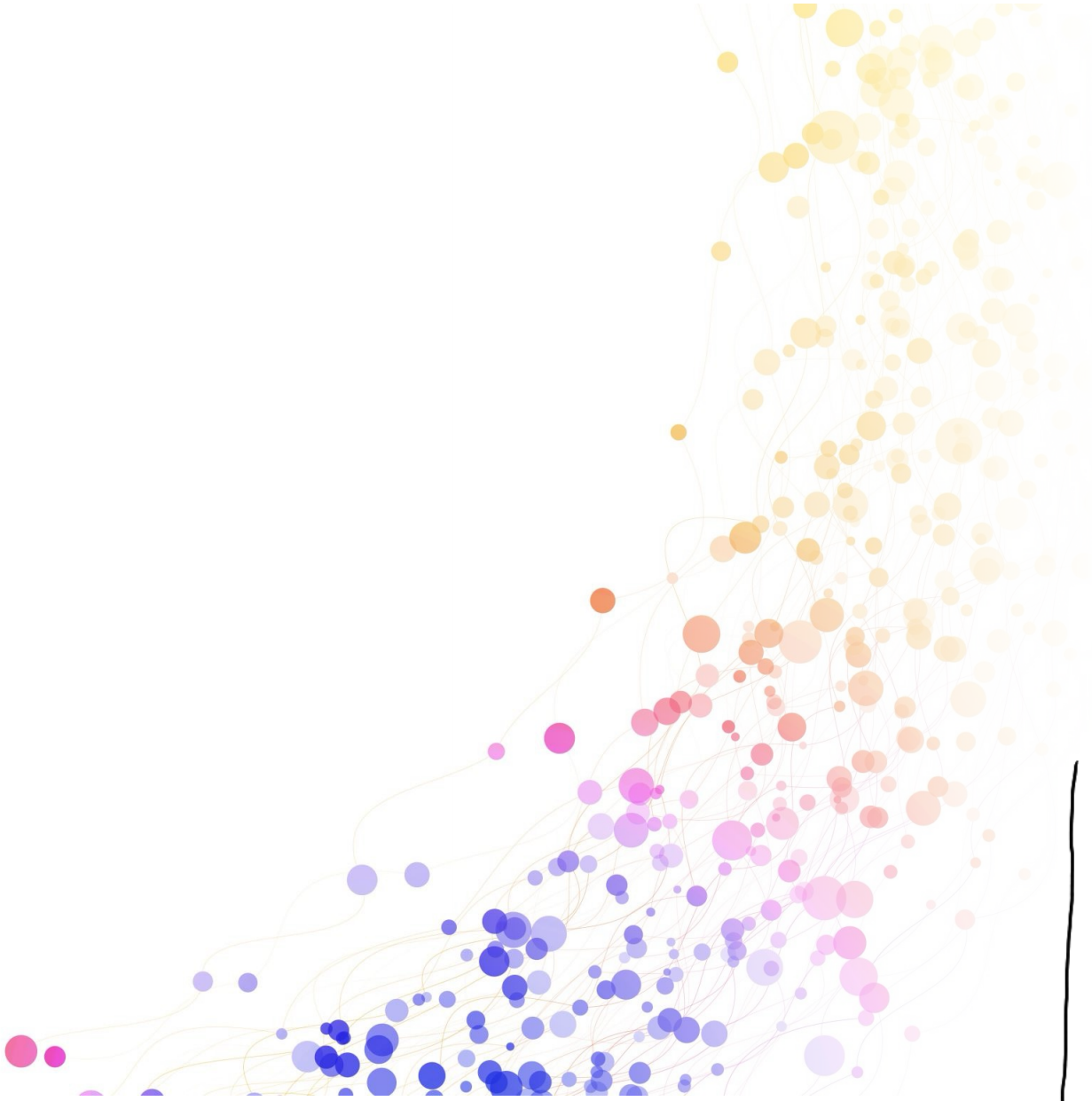
```
x := 0;
y := 0;
while (n > 0) {
  if (n % 2 = 0) {
    x := x + n;
    y := y + 1;
  }
  else {
    y := y + n;
    x := x + 1;
 }
  n := n - 1;
}
print(x);
```



Why go through all this trouble?

- ✓ Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
- ✓ We may want a compiler with multiple front ends for multiple languages; rather than translating each language to a CFG, translate each language to a canonical IR and then to a CFG

- Why? They are closer to intermediate representation. We can perform SA and CFG can be easily translated into intermediate representation (by recode)

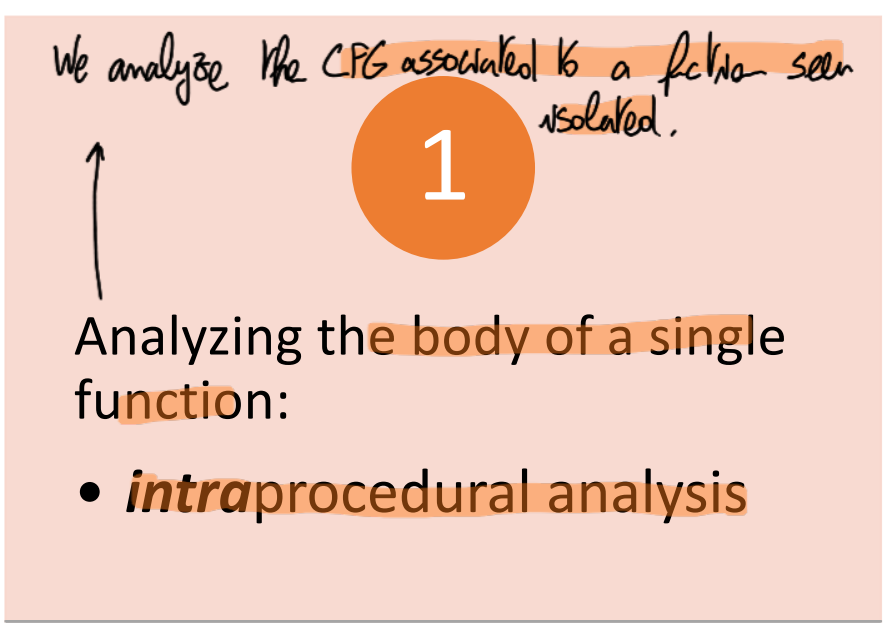


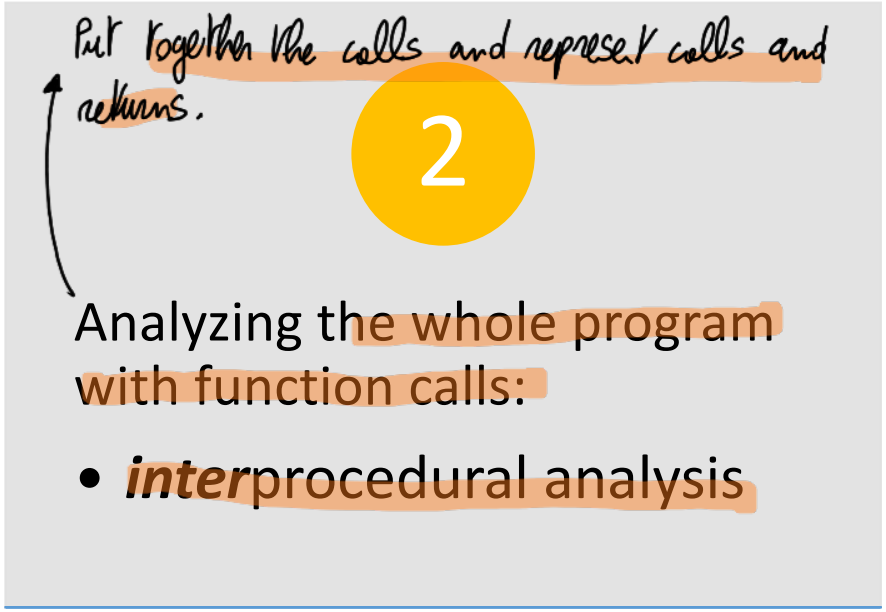
ANALYZING CFGs

We can associate constraints to basic blocks, not to program points. Fewer constraints. No longer code base problem. But how to analyze CFG? How to manage functions? We will see the C strategy. ①

① All functions are global and no local nested function declarations.

Two issues


We analyze the CPG associated to a function seen isolated.
1
Analyzing the body of a single function:
• intraprocedural analysis


Put together the calls and represent calls and returns.
2
Analyzing the whole program with function calls:
• interprocedural analysis

CFG for whole programs

Construct a CFG for each function

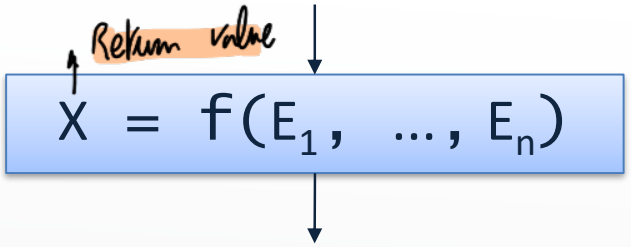
Then glue them together to reflect function calls and returns

We need to take care of:

- Parameter passing
- Return values
- Values of local variables across calls (including recursive functions, so not enough to assume unique variable names)

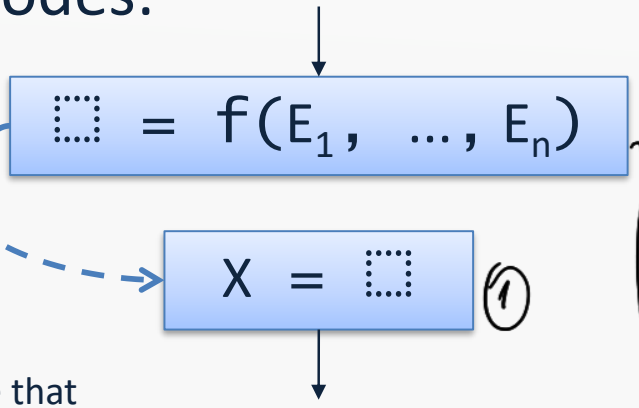

FUNCTION:
The calling convention
Prologue + epilogue
• Prologue evaluates parameters and pushes activation record on stack.
• Epilogue pops, fixes return address and takes care of return values.

Split each original call node



function call is represented like this.
Block expressing call with parameters and result to be placed in X.

into two nodes:



the "call node"

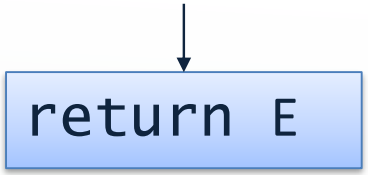
the "after-call node"

We split calling node in two:
1: The call node representing call (flow from caller to callee. Represent evaluation of parameters done in environment of caller (call by value)

a special edge that connects the call node with its after-call node

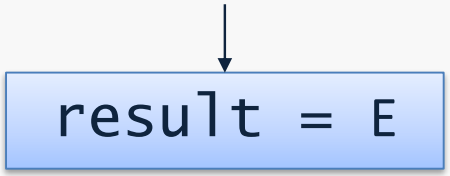
(1) Epilogue: we put a box that represent the value computed that will be saved in X. We simulate prologue and epilogue.

Change each return node



In CFG return is a node like this.
We represent it like result = E.

into an assignment:

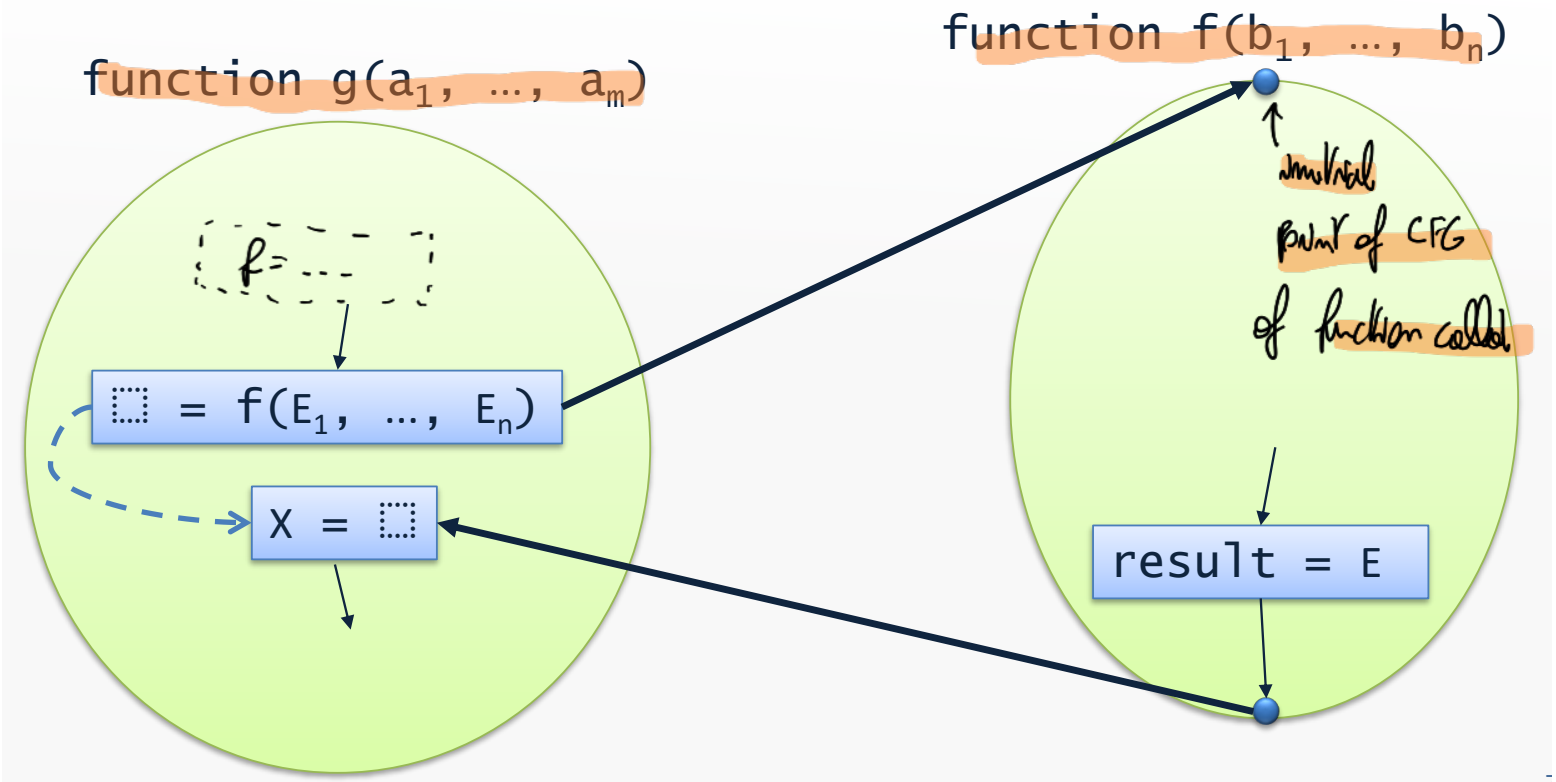


(where `result` is a fresh variable)

↳ fresh: we might have several results and we have to identify them.
FRESH: no other possible existing variables.

A unique variable generated by compiler.

Add call edges and return edges:



If we have 2 calls, we need another arrow for beginning and for result. This creates a problem. In analysis you might confuse returns and calls.

CFG: Summary

one notion to represent sequential programs but also function calls.

The control flow graph shows all the paths that can be traversed during a program execution.

- A control flow graph is a directed graph.
- Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks.

Advantages of CFG

- They can easily encapsulate the information per each basic block.
- They can easily locate inaccessible codes of a program and syntactic structures such as loops are easy to find in a control flow graph.

And in analysis we can use blocks to our advantage


It’s time to go back to
static analysis

Static Analysis: The ingredients

The CFG of the program to be analyzed

Constraint generation

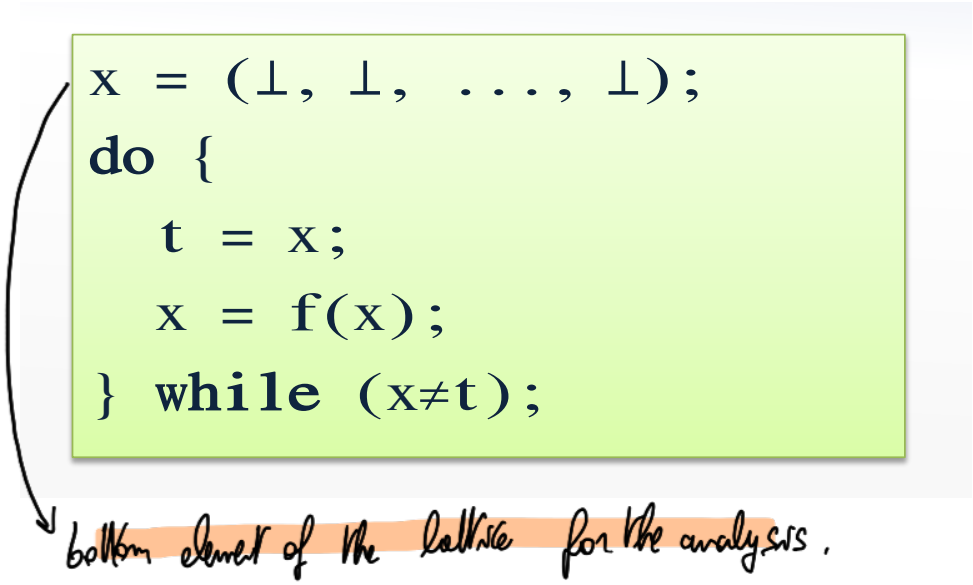
- For each node v of the CFG we introduce the constraint variable [v]
- For eack cobstraint variable we derive the data flow constraints

Constraint Solving

- Monotone framework and fixpoint

SA takes CFG of program (back end of compiler) → We generate the constraints for each node of the graph. And then we solve the constraints through fixed points and monotone framework. [v]: constraints associated to basic block

Constraint solving: the naïve algorithm



We start with no information $(\bot, \ldots \bot)$ and iterate to pickup better and better solutions until $f^{k+1}(\bot) = f^k(\bot)$

Constraint solving: the naïve algorithm

```
x = (⊥, ⊥, ..., ⊥);
do {
   t = x;
   x = f(x);
} while (x≠t);
```

Correctness of the algorithm is ensured by the monotone framework (the fixed point theorem)

Well defined hypothesis $f(x_1,...,x_n) = (f_1(x_1,...,x_n), ..., f_n(x_1,...,x_n))$

We can represent it this way

The Naïve Algorithm

We apply them until value in kth column is equal to (k-1)th. We then stop.

CONSTRAINTS

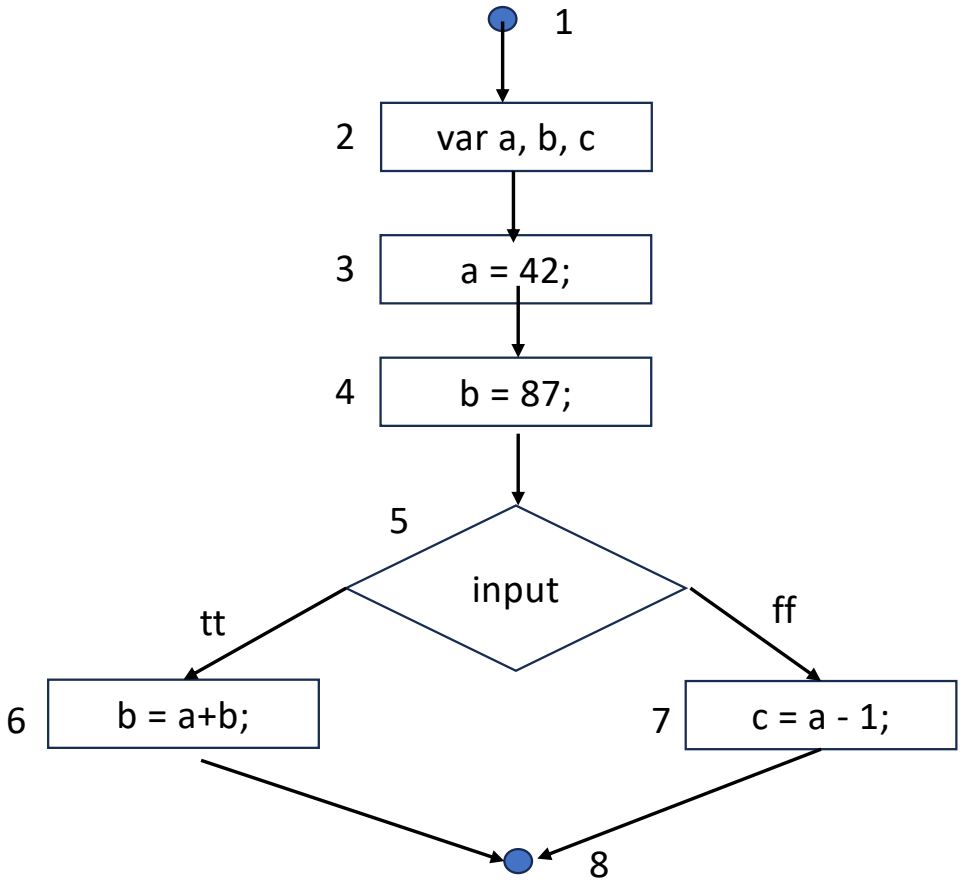
	$f^0(\bot, \bot, \ldots, \bot)$	$f^1(\bot, \bot, \ldots, \bot)$	...	$f^k(\bot, \bot, \ldots, \bot)$
1	$\bot$	$f_1(\bot, \bot, \ldots, \bot)$	...	...
2	$\bot$	$f_2(\bot, \bot, \ldots, \bot)$	...	...
...	...	...	...	...
n	$\bot$	$f_n(\bot, \bot, \ldots, \bot)$	...	...

Computing each new entry is done using the previous column

Without using the entries in the current column that have already been computed

Many entries are likely unchanged from one column to the next!

SIGN Analysis (again)



$$X_1 = \bot$$

$$X_2 = X_1[a \to \top, b \to \top, c \to \top]$$

$$X_3 = X_2[a \to +]$$

$$X_4 = X_3[b \to +]$$

$$X_5 = X_4$$

$$X_6 = X_5\,[c \to \big(eval(X_5, a + b)\big)]$$

$$X_7 = X_5[c \to eval(X_5, a\ - b))]$$

$$X_8 = X_6 \sqcup X_7$$

The Naïve Algorithm

	$f^0(\bot, \ldots, \bot)$	$f^1(\bot, \ldots, \bot)$	.....	$f^k(\bot, \ldots, \bot)$
1	$\bot$	$\bot$		
2	$\bot$	$[a \to \top, b \to \top, c \to \top]$		
3	$\bot$	$[a \to +]$		
4	$\bot$	$[b \to +]$		
5	$\bot$	$\bot$		
6	$\bot$	$[c \to \bot]$		
7	$\bot$	$[c \to \bot]$		
8	$\bot$	$[c \to \bot]$		

8

Can we do better?

CHAOTIC ITERATION

We can do better: chaotic iteration for fixed point iteration

$x_1 = \bot; \ \ldots \ x_n = \bot;$

while $((x_1, \ldots, x_n) \neq f(x_1, \ldots, x_n))$ {

pick **i** nondeterministically such that $x_i \neq f_i(x_1, \ldots, x_n)$

$x_i = f_i(x_1, \ldots, x_n);$

}

We continue to do iteration while current approx is different from the following one

We replace values

Take a random constraint and make it evolve !

We apply function locally for dimension i, What might happen is that

Constraint is something like:

$$X_n = f(X_1, \ldots X_m)$$

- Inside function you can associate nodes in the right hand side of equations. You find dependencies of a node, what happens on the right hand side of the constraint.
- You get X_n and apply f_n to get a new value X_n for next iteration

Are you capable to understand nodes involved in the execution of a function? Yes. Modifications of X_n depends on X_I, X_K...

The worklist algorithm

A specialization of chaotic iteration that exploits the special structure of the function f to be computed

Key Observation1: Most right-hand sides of the function components f_i are quite sparse

Key Observation 2: Constraints on CFG nodes do not involve all others

The worklist algorithm: auxiliary function

Use a map:

$$dep: \text{Nodes} \rightarrow 2^{\text{Nodes}}$$

that for $v \in \text{Nodes}$ gives the set of nodes (i.e. constraint variables) w where v occurs on the right-hand side of the constraint for w

The worklist algorithm

start with a set of nodes, remove one non deterministically and check value ①

```
x1 = ⊥; ... xn = ⊥;
W = {v1, ..., vn};
while (W≠∅) {
   vi = W.removeNext();
   y = fi(x1, ..., xn);
   if (y≠xi) {
      for (vj ∈ dep(vi)) W.add(vj);
      xi = y;
   }
}
```

Nodes involved in the constraints.

If $W \neq \emptyset$ you still have nodes to analyze.

- $dep(v_i)$: nodes you find in the right hand side.

① But you insert in the set W only nodes involved, not all the nodes. Other nodes are left unchanged. When set is empty you have finished

OH! When we talk about dependencies of vi, we mean the nodes that depend on vi, not the list of nodes on which vi depends from?

Exactly! 🎯

When we say "**dependencies of v_i**" in this context, we actually mean **the nodes that depend on v_i**—not the nodes that v_i depends on.

This is a bit counterintuitive because "dependencies" often means "things that I rely on," but here we mean "things that rely on me."

So, if we have:

$$x_2 = f_2(x_1)$$

- x_2 **depends on** x_1 (it needs x_1 to compute its value).
- **But in the algorithm,** when x_1 changes, we put v_2 (corresponding to x_2) in the worklist—**because x_2 depends on x_1 and might need to change**.

I assume if you only have V_i that is evaluated once then removed from W and not added anymore, it means it does not depend from other nodes, hence it is constant, only need one update: $X_i = f(X_1, \dots X_m) = c$. Will not be put back in W.

READING NOTE AVAILABLE ON TEAMS

READING MATERIAL

LECTURE 8 March 2024