Lecture 10: Dataflow/Structural Analysis

COS 598C – Advanced Compilers

Bolei Guo

Prof. David August
Department of Computer Science
Princeton University

Where are we?

- Analysis
  - Control Flow/Predicate
    - Treat basic blocks as a black box
    - Only look at branches
  - Dataflow
    - Look inside basic blocks
    - What is computed where?

Basic Block Level Analysis

To improve performance of dataflow, process at basic block level.

- Represent the entire basic block by a single super-instruction which has any number of destinations and sources.
- Run dataflow at basic block level.
- Expand result to the instruction level.

Example:

```
p:  r1 = r2 + r3     ->   r1, r2 = r2, r3
n:  r2 = r1
```


Basic Block Level Analysis

- Example:

```
p:  r1 = r2 + r3     ->   r1, r2 = r2, r3
n:  r2 = r1
```

- For reading definitions:

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$

But $IN[n] = OUT[p]$:

$$OUT[n] = GEN[n] \cup ((GEN[p] \cup (IN[p] - KILL[p])) - KILL[n])$$

Which (clearly) yields:

$$OUT[n] = GEN[n] \cup (GEN[p] - KILL[n]) \cup (IN[p] - (KILL[p] \cup KILL[n]))$$

So:

$$GEN[pn] = GEN[n] \cup (GEN[p] - KILL[n])$$
$$KILL[pn] = KILL[p] \cup KILL[n]$$

- Can we do this at the loop or general region level?

Other Regions

- Lists of instructions - Basic Blocks!

$$GEN[pn] = GEN[n] \cup (GEN[p] - KILL[n])$$
$$KILL[pn] = KILL[p] \cup KILL[n]$$

- Conditionals/Hammocks

$$GEN[lr] = GEN[l] \cup GEN[r]$$
$$KILL[lr] = KILL[l] \cap KILL[r]$$

- While Loops

$$GEN[loop] = GEN[l]$$
$$KILL[loop] = KILL[l]$$

Try this on an irreducible flow graph...

Two approaches to control & data flow analysis

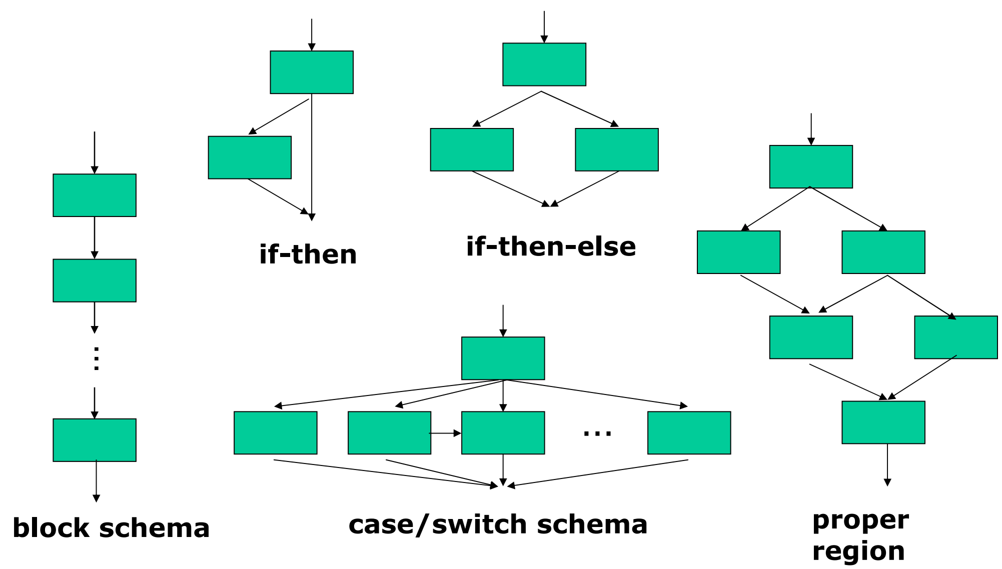
Iterative analysis
- Construct CFG
- Compute transfer function for each node
- Solve the dataflow equations by iterating over the CFG

Structural analysis
- Decompose CFG into nested control structures
- Compute transfer function for each control structure
- Propagate dataflow information into and through the control structures starting from the top-level control structure

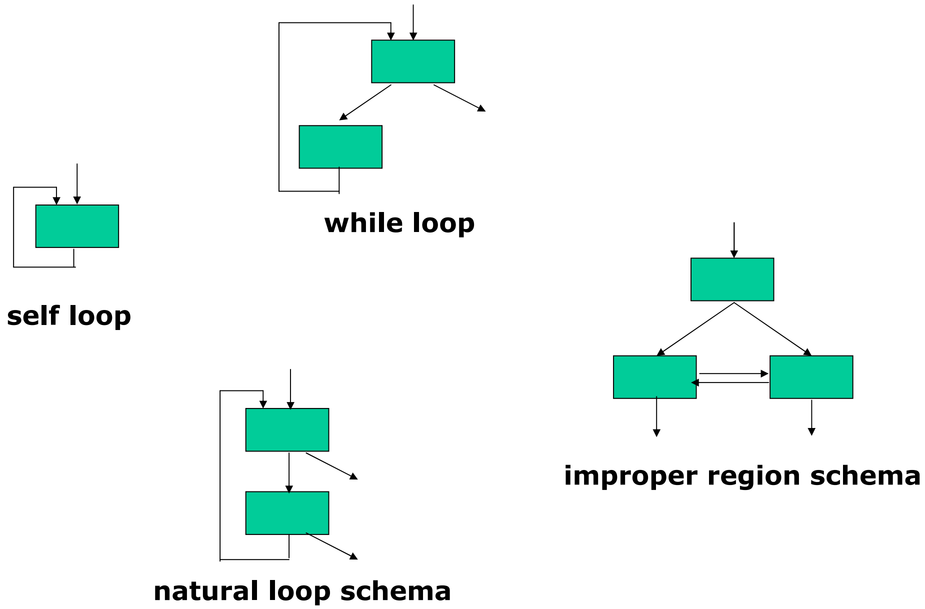
Why structural analysis?

- The actual dataflow analysis is faster
- It's easier to update dataflow information incrementally
- Makes control-flow transformations easier

Classification of control structures — Acyclic regions



Classification of control structures — Cyclic regions



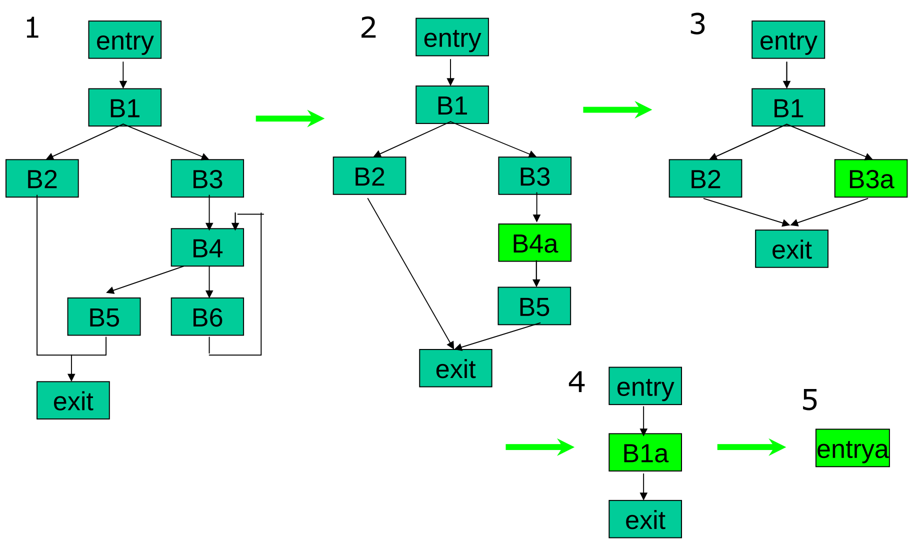
An important property of the regions

- Single-entry
- Improper regions always include the lowest common dominator of all the entries of its multi-entry strongly-connected component.

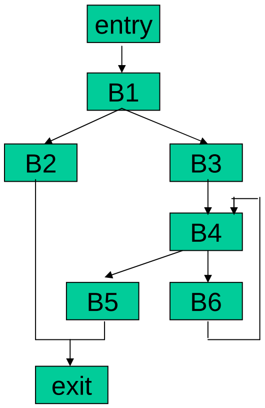
Flowgraph reduction

- Collapse each control structure into an abstract node, the resulting flowgraph is an abstract flowgraph.
- Apply reductions to the abstract flowgraph, the resulting regions are nested.
- Control tree:
  - *Leaves* – basic blocks
  - *Root* – an abstract graph corresponding to the original cfg
  - *Internal nodes* – abstract nodes each corresponding to a subgraph of the original cfg

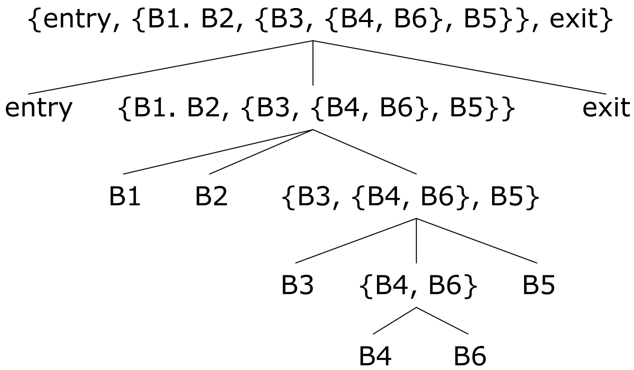
Flowgraph reduction example 1



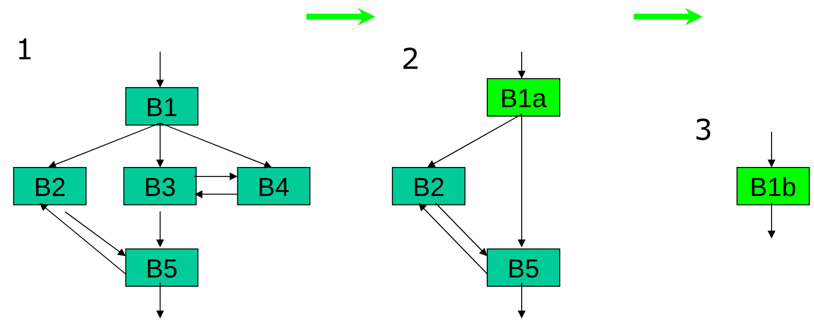
Flowgraph reduction example 1



Control tree



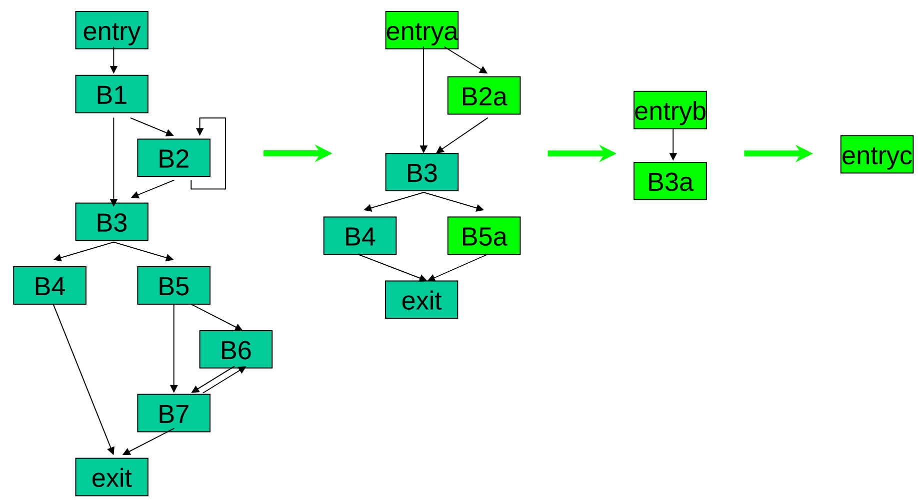
Flowgraph reduction example 2



Flowgraph reduction algorithm

```
structural_analysis(G) {
    repeat
        for (n : DFS_Postorder(G))
            if (n is in an acyclic region)
                reduce the region
            else
                C = {n}
                for each node m
                    if (∃path m~>k->n &&
                        k->n is a back edge)
                        C ∪= {m}
                if (C is a cyclic region)
                    reduce C
    until G is reduced to a single node
}
```


Flowgraph reduction class problem



Structural dataflow analysis

2 passes over the control tree

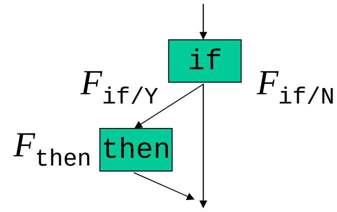
Bottom-up pass:

- Construct a transfer function for each node

Top-down pass:

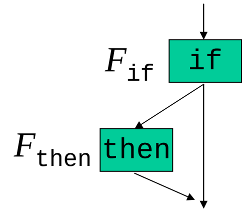
- Construct and evaluate dataflow equations that propagate initial dataflow information into and through each node, using the functions constructed in the first pass

"if-then" construct — bottom-up pass



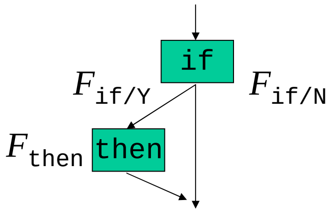
$$F_{if-then} = (F_{then} \circ F_{if/Y}) \wedge F_{if/N}$$

This is more precise if dataflow values are different along the two branches, e.g. constant propagation.



$$F_{if-then} = (F_{then} \circ F_{if}) \wedge F_{if}$$

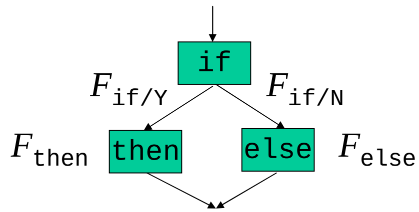
"if-then" construct — top-down pass



$$in_{if} = in_{if-then}$$

$$in_{then} = F_{if/Y}(in_{if})$$

"if-then-else" construct



$$F_{if-then-else} =$$

$$(F_{then} \circ F_{if/Y}) \wedge (F_{else} \circ F_{if/N})$$

$$in_{if} = in_{if-then-else}$$

$$in_{then} = F_{if/Y}(in_{if})$$

$$in_{else} = F_{if/N}(in_{if})$$

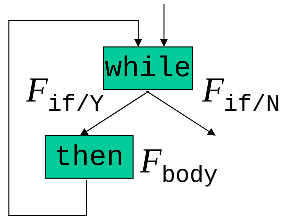
General acyclic region $A = \{B0, B1, \ldots, Bn\}$

- B0 is the entry node
- Each Bi has exits $Bi/1, \ldots, Bi/e_i$ with transfer functions $F_{Bi/1}, \ldots, F_{Bi/e_i}$
- For some exit Bi_k/e_k, let $P(A, Bi_k/e_k)$ denote the set of all possible paths from the entry of A to it, the transfer function for these paths is

$$F_{(A,\, Bi_k/e_k)} = \bigwedge_{p \in P(A,\, Bi_k/e_k)} F_p$$

- For any $p = B0/e_0, Bi_1/e_1, \ldots, Bi_k/e_k \in P(A, Bi_k/e_k)$, $F_p = F_{Bi_k/e_i} \circ \cdots \circ F_{Bi_1/e_1} \circ F_{B0/e_0}$

while-loop



$$F_{while-loop} = F_{while/N} \circ F_{iter}^{\ *}$$
$$= F_{while/N} \circ (F_{body} \circ F_{while/Y})^*$$
$$in_{while} = F_{iter}^{\ *}(in_{while-loop})$$
$$= (F_{body} \circ F_{while/Y})^*(in_{while-loop})$$
$$in_{body} = F_{while/Y}(in_{while})$$

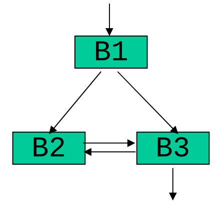
Proper cyclic region $C = \{B0, B1, \ldots, Bn\}$

- There is a single back edge $(Bc/e, B0)$
- In the acyclic region resulting from removing the back edge, construct a transfer function $F'_{(C, Bi_k/e_k)}$ that corresponds to all possible paths from C's entry to each exit Bi_k/e_k
- The transfer function for executing C and exiting from Bi_k/e_k is

$$F_{(C,\, Bi_k/e_k)} = F'_{(C,\, Bi_k/e_k)} \circ F_{iter}^{\ *}$$
$$= F'_{(C,\, Bi_k/e_k)} \circ F'^{\ *}_{(C, Bc/e)}$$

Improper region

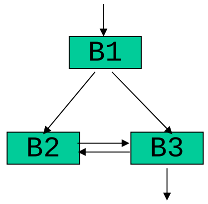
- Bottom-up pass - the same as acyclic regions
- Top-down pass – the equations are recursive



$$F_{B1-B2-B3} = \left((F_{B3} \circ F_{B2})^+ \wedge ((F_{B3} \circ F_{B2})^* \circ F_{B3})\right) \circ F_{B1}$$
$$in_{B1} = in_{B1-B2-B3}$$
$$in_{B2} = F_{B1}(in_{B1}) \wedge F_{B3}(in_{B3})$$
$$in_{B3} = F_{B1}(in_{B1}) \wedge F_{B2}(in_{B2})$$

3 ways to deal with recursive equations

- Turn the improper region into a proper one using node splitting.
- Evaluate the recursive equations together iteratively.
- For many dataflow problems, non-recursive transfer functions can be computed.



$$in_{B2} = \left((F_{B3} \circ F_{B2})^{*} \circ ((F_{B3} \circ F_{B1}) \wedge F_{B1}) \right) (in_{B1})$$
$$= \left(((F_{B3} \circ F_{B2}) \wedge id) \circ ((F_{B3} \circ F_{B1}) \wedge F_{B1}) \right) (in_{B1})$$
$$in_{B3} = \left((F_{B3} \circ F_{B2})^{*} \circ ((F_{B2} \circ F_{B1}) \wedge F_{B1}) \right) (in_{B1})$$
$$= \left(((F_{B3} \circ F_{B2}) \wedge id) \circ ((F_{B2} \circ F_{B1}) \wedge F_{B1}) \right) (in_{B1})$$

Reducible Flow Graphs

Definition

- A flow graph is reducible iff each edge exists in exactly one class:
  1. Forward edges (forms an acyclic graph where every node is reachable from start node)
  2. Back edges (head dominates tail)

Algorithm:

1. Remove all backedges
2. Check for cycles:
   - Cycles: Irreducible.
   - No Cycles: Reducible.

Think:

- All loop entry arcs point to header.

Reducible Flow Graphs – Structured Programs

Motivation:

- Structured programs are always reducible programs.
- Reducible programs are not always structured programs.
- Exploit the structured or reducible property in dataflow analysis.

Structures:

- Lists of instructions
- Conditionals/Hammocks
- While Loops (no breaks)