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: \( r1 = r2 + r3 \)  \( \rightarrow \) \( r1, r2 = r2, r3 \)

n: \( r2 = r1 \)
Basic Block Level Analysis

- Example:
  p: \( r_1 = r_2 + r_3 \) \( \rightarrow \) \( r_1, r_2 = r_2, r_3 \)
  n: \( r_2 = r_1 \)

- For reaching 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!
  \( GEN[bn] = GEN[n] \cup (GEN[p] - KILL[n]) \)
  \( KILL[bn] = KILL[p] \cup KILL[n] \)

- Conditionals/Hammocks
  \( GEN[tr] = GEN[t] \cup GEN[r] \)
  \( KILL[tr] = KILL[t] \cap KILL[r] \)

- 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

- Block schema
- Case/switch schema
- Proper region

Classification of control structures — Cyclic regions

- While loop
- Improper region schema
- Natural loop schema
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

Flowgraph reduction example 2

Flowgraph reduction algorithm

```plaintext
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 (exists 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

\[ F_{if-then} = (F_{then} \circ F_{if/Y}) \land F_{if/N} \]
This is more precise if dataflow values are different along the two
branches, e.g. constant propagation.

\[ F_{if-then} = (F_{then} \circ F_{if}) \land F_{if} \]
“if-then” construct — top-down pass

\[
\begin{align*}
F_{\text{if/Y}} & \quad \text{if} \quad F_{\text{if/N}} \\
F_{\text{then}} & \quad \text{then} \\
in_{\text{if}} & = in_{\text{if-then}} \\
in_{\text{then}} & = F_{\text{if/Y}}(in_{\text{if}})
\end{align*}
\]

“if-then-else” construct

\[
\begin{align*}
F_{\text{if/Y}} & \quad \text{if} \quad F_{\text{if/N}} \\
F_{\text{then}} & \quad \text{then} \\
in_{\text{if}} & = in_{\text{if-then-else}} \\
in_{\text{then}} & = F_{\text{if/Y}}(in_{\text{if}}) \\
in_{\text{else}} & = F_{\text{if/N}}(in_{\text{if}})
\end{align*}
\]

**General acyclic region** \( A = \{B_0, B_1, \ldots, B_n\} \)

- \(B_0\) is the entry node
- Each \(B_i\) has exits \(B_i/1, \ldots, B_i/e_j\) with transfer functions \(F_{B_i/1}, \ldots, F_{B_i/e_j}\)
- For some exit \(B_{i_k}/e_k\), let \(P(A, B_{i_k}/e_k)\) denote the set of all possible paths from the entry of \(A\) to it, the transfer function for these paths is
  \[
  F_{(A, B_{i_k}/e_k)} = \bigwedge_{p \in P(A, B_{i_k}/e_k)} F_p
  \]
- For any \(p = B_0/e_0, B_{i_j}/e_j, \ldots, B_{i_k}/e_k \in P(A, B_{i_k}/e_k)\),
  \[
  F_p = F_{B_{i_j}/e_j} \circ \cdots \circ F_{B_{i_k}/e_k} \circ F_{B_0/e_0}
  \]
while-loop

\[
F_{\text{while-loop}} = F_{\text{while}/N} \circ F_{\text{iter}}^* = F_{\text{while}/N} \circ (F_{\text{body}} \circ F_{\text{while}/Y})^* \\
\text{in}_{\text{while}} = F_{\text{iter}}^* \text{(in}_{\text{while-loop})} \\quad = (F_{\text{body}} \circ F_{\text{while}/Y})^* \text{(in}_{\text{while-loop})} \\
\text{in}_{\text{body}} = F_{\text{while}/Y} \text{(in}_{\text{while})}
\]

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 \( F'(C, Bk/e_k) \) that corresponds to all possible paths from \( C \)'s entry to each exit \( Bk/e_k \)
- The transfer function for executing \( C \) and exiting from \( Bk/e_k \) is
  \[
  F(C, Bk/e_k) = F'(C, Bk/e_k) \circ F_{\text{iter}}^* = F'(C, Bk/e_k) \circ F'(C, Bc/e)^*
  \]

Improper region

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

\[
F_{B1-B2-B3} = (F_{B3} \circ F_{B2})^+ \land ((F_{B3} \circ F_{B2})^* \circ F_{B3}) \circ F_{B1} \\
\text{in}_{B1} = \text{in}_{B1-B2-B3} \\
\text{in}_{B2} = F_{B1}(\text{in}_{B1}) \land F_{B3}(\text{in}_{B3}) \\
\text{in}_{B3} = F_{B1}(\text{in}_{B1}) \land F_{B2}(\text{in}_{B2})
\]
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.

\[
in_{B_2} = \left( (F_{B_3} \circ F_{B_2})^* \circ ((F_{B_3} \circ F_{B_1}) \land F_{B_1}) \right)(in_{B_1})
\]

\[
= \left( ((F_{B_3} \circ F_{B_2}) \land id) \circ ((F_{B_3} \circ F_{B_1}) \land F_{B_1}) \right)(in_{B_1})
\]

\[
in_{B_3} = \left( (F_{B_3} \circ F_{B_2})^* \circ ((F_{B_3} \circ F_{B_1}) \land F_{B_1}) \right)(in_{B_1})
\]

\[
= \left( ((F_{B_3} \circ F_{B_2}) \land id) \circ ((F_{B_2} \circ F_{B_1}) \land F_{B_1}) \right)(in_{B_1})
\]

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)