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:

\[ p: \quad r_1 = r_2 + r_3 \quad \rightarrow \quad r_1, r_2 = r_2, r_3 \]
\[ n: \quad r_2 = r_1 \]

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

- Example:
  - \[ p: \quad r_1 = r_2 + r_3 \quad \rightarrow \quad r_1, r_2 = r_2, r_3 \]
  - \[ n: \quad r_2 = r_1 \]

- For reaching definitions:
  - \[ \text{OUT}[n] = \text{GEN}[n] \cup (\text{IN}[n] - \text{KILL}[n]) \]
  - But \[ \text{IN}[n] = \text{OUT}[p] \]:
    - \[ \text{OUT}[n] = \text{GEN}[n] \cup ((\text{GEN}[p] \cup (\text{IN}[p] - \text{KILL}[p])) - \text{KILL}[n]) \]
  - Which (clearly) yields:
    - \[ \text{OUT}[n] = \text{GEN}[n] \cup (\text{GEN}[p] - \text{KILL}[n]) \cup (\text{IN}[p] - (\text{KILL}[p] \cup \text{KILL}[n])) \]
  - So:
    - \[ \text{GEN}[pm] = \text{GEN}[n] \cup (\text{GEN}[p] - \text{KILL}[n]) \]
    - \[ \text{KILL}[pm] = \text{KILL}[p] \cup \text{KILL}[n] \]

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

---

- 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...
Classification of control structures — Cyclic regions

- While loop
- Self 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

1. entry
   - B1
     - B2
     - B3
     - B5
     - exit
2. entry
   - B1
     - B2
     - B4
     - B5
     - exit
3. entry
   - B1
     - B2
     - B3
     - B4a
     - B5
     - exit
4. entry
   - B1a
     - B2
     - exit
5. entry
   - B1a
     - entrya
     - exit
Flowgraph reduction example 1

Flowgraph reduction example 2

Control tree

```
{entry, {B1, B2, {B3, {B4, B6}, B5}}, exit}
```

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

\[ F_{\text{if-then}} = (F_{\text{then}} \circ F_{\text{if}/Y}) \land F_{\text{if}/N} \]

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

“if-then” construct — top-down pass

\[ F_{\text{if}} = F_{\text{if}/Y}(\text{in}_f) \]

\[ \text{in}_{\text{then}} = F_{\text{if}/Y}(\text{in}_f) \]

“if-then-else” construct

\[ F_{\text{if-then-else}} = (F_{\text{then}} \circ F_{\text{if}/Y}) \land (F_{\text{else}} \circ F_{\text{if}/N}) \]

\[ \text{in}_{\text{if}} = \text{in}_{\text{if-then-else}} \]

\[ \text{in}_{\text{then}} = F_{\text{if}/Y}(\text{in}_f) \]

\[ \text{in}_{\text{else}} = F_{\text{if}/N}(\text{in}_f) \]
**General acyclic region**  \( A = \{B_0, B_1, \ldots, B_n\} \)

- B0 is the entry node
- Each \( B_i \) has exits \( B_i/1, \ldots, B_i/e \) with transfer functions \( F_{B_i/1}, \ldots, F_{B_i/e} \)
- For some exit \( B_i/k \), let \( P(A, B_i/k) \) denote the set of all possible paths from the entry of A to it, the transfer function for these paths is
  \[
  F_{(A, B_i/k)} = \bigwedge_{p \in P(A, B_i/k)} F_p
  \]
- For any \( p = B_0/e_0, B_i/e_1, \ldots, B_i/k \in P(A, B_i/k) \),
  \[
  F_p = F_{B_i/e_0} \circ \cdots \circ F_{B_i/e_1} \circ F_{B_0/e_0}
  \]

**while-loop**

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

**Proper cyclic region**  \( C = \{B_0, B_1, \ldots, B_n\} \)

- There is a single back edge \((B_c/e, B_0)\)
- In the acyclic region resulting from removing the back edge, construct a transfer function \( F'_{(C, B_i/k)} \) that corresponds to all possible paths from \( C \)'s entry to each exit \( B_i/k \)
- The transfer function for executing \( C \) and exiting from \( B_i/k \) is
  \[
  F_{(C, B_i/k)} = F'_{(C, B_i/k)} \circ F_{\text{iter}}^*
  \]
  \[
  = F'_{(C, B_i/k)} \circ F'_{(C, B_c/e)}^*
  \]

**Improper region**

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

\[
F_{B_1-B_2-B_3} = ((F_{B_3} \circ F_{B_2}^+) \land ((F_{B_3} \circ F_{B_2}^* \circ F_{B_3})) \circ F_{B_1}
\]
\[
in_{B_1} = in_{B_1-B_2-B_3}
\]
\[
in_{B_2} = F_{B_1}(in_{B_1}) \land F_{B_3}(in_{B_3})
\]
\[
in_{B_3} = F_{B_1}(in_{B_1}) \land F_{B_2}(in_{B_2})
\]
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.

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

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

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

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

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)