Proper Ordering for Nodes

Flow-Graph Depth
Loop Structure of Reducible Flow Graphs

Jeffrey D. Ullman
Stanford University
1. Proper ordering of nodes during iterative algorithm assures number of passes limited by the number of “nested” back edges.
2. Depth of nested loops upper-bounds the number of nested back edges.
Suppose that for a RD analysis, we visit nodes during each iteration in DF order.
The fact that a definition $d$ reaches a block will propagate in one pass along any sequence of blocks whose DF numbers increase.
- I.e., $d$ propagates along any edge that is not retreating.

When $d$ arrives at the tail of a retreating edge, it is too late to propagate $d$ from OUT to IN.
- The IN at the head has already been computed for that round.
Definition d is Gen’d by node 2 not killed anywhere.

The first pass

The second pass
The depth of a flow graph with a given DFST and DF-order is the greatest number of retreating edges along any acyclic path.

For RD, if we use DF order to visit nodes, we converge in depth+2 passes.

- Depth+1 passes to follow that number of increasing segments.
- 1 more pass to realize we converged.
  - Or figure out the depth in advance, so you know you converged.
Example: Depth = 2

- Increasing, Retreating, Increasing
  - Pass 1
- Increasing, Retreating, Increasing
  - Pass 2
- Increasing
  - Pass 3
Similarly . . .

- AE also works in depth+2 passes.
  - Unavailability propagates along retreat-free node sequences in one pass.
- So does LV if we use reverse of DF order.
  - A use propagates backward along paths that do not use a retreating edge in one pass.
In General . . .

- The depth+2 bound works for any monotone framework, as long as information only needs to propagate along acyclic paths.
  - Example: if a definition reaches a point, it does so along an acyclic path.
Normal control-flow constructs produce reducible flow graphs with the number of back edges at most the nesting depth of loops.

- Nesting depth tends to be small.
Example: Nested Loops

3 nested while-loops; depth = 3.

while(...) do
  while(...) do
    while(...) do
      something

3 nested repeat-loops; depth = 1

repeat
  repeat
    repeat
      something
    until
  until
  until
The natural loop of a back edge $a \rightarrow b$ is $b$ plus the set of nodes that can reach $a$ without going through $b$.

Theorem: two natural loops are either disjoint, identical, or one is contained (nested) within the other.

Thus, for reducible flow graphs, the nesting structure of loops is well defined.
Example: Natural Loops

Natural loop of 5 -> 1

Natural loop of 3 -> 2
Assume we only need to propagate along acyclic paths.

If a path reaches the header of a natural loop, we can follow no back edges within that region.

- **Why?** Within a natural loop, you must reach the head of a back edge before the tail (Shown on next slide).

Thus, any sequence of back edges must be to headers of loops, each of which contains the previous loop.

That is why the nesting depth of loops is an upper bound on the depth of a reducible flow graph.
h dominates v and w.

v dominates w

Acyclic path reaches h, then v before it reaches w. So we cannot follow back edge v->w, as it would complete a cycle.
Region-Based Analysis

$T_1$ and $T_2$ Definition of Reducibility Regions
Transfer Functions Within Regions

Jeffrey D. Ullman
Stanford University
The iterative approach to DFA is simple and efficient.

Why would we want to find another approach – one that focuses on the loop structure of reducible flow graphs?

The structured approach can be a bit more efficient.

More importantly, it is almost essential when we do things that depend on the loop structure.

Example: induction variables.
A set of nodes $N$ and edges $E$ is a region if:

1. There is a header $h$ in $N$ that dominates all nodes in $N$.
2. If $n \neq h$ is in $N$, then all predecessors of $n$ are also in $N$.
   - I.e., the header is the sole entry into the region.
3. $E$ consists of all edges between nodes in $N$, possibly excluding nodes that enter $h$ from somewhere in $N$.

**Note:** Every node is a region by itself.
For reducible flow graphs, we can “reduce” the graph by two region-creating transformations.

- **T1**: Remove a self-loop from a node.
- **T2**: combine two nodes \( n \) and \( m \), where \( m \)’s only predecessor is \( n \), and \( m \) is not the entry.
  - Say “\( n \) consumes \( m \)”
Example: T1-T2 Reduction

Note edge 3->2 is excluded
Example: $T_1$-$T_2$ Reduction
Example: T1-T2 Reduction
Example: \textbf{T}_1-\textbf{T}_2 Reduction
Example: \( T_1 - T_2 \) Reduction
Example: $T_1$-$T_2$ Reduction
Regions Constructed During T1-T2 Reduction

- As we reduce, each node represents a region of the original flow graph, and each edge represents one or more edges of the original flow graph.
- **T2**: Take the union of the two regions represented by the two nodes, plus all edges represented by the edge from the consumer to the consumed node.
- **T1**: Add (to the region represented by the node) those edges represented by the loop that was removed.
- **Easy inductive proof:** every set of nodes and edges constructed is a region.
  - **Key point:** When you use T2, the header of the resulting region is the header of the region that consumes the other.
Example: T₁-T₂ Reduction
- If $R$ is a region with header $h$, then for all blocks $B$ in $R$ with an out-edge not in $R$: $f_{R, \text{OUT}[B]} = \text{meet}$ over paths in $R$ from the beginning of the header of $R$ to end of $B$.

- For any region $R$: $f_R = \text{meet}$ over paths from the beginning of the header of $R$ to itself.

- If $T$ is a region constructed by $R$ consuming $S$: $f_{T,S} = \text{meet}$ over paths in $R$ from the beginning of the header of $T$ (and of $R$) to the beginning of the header of $S$. 

Transfer Functions
Note convention: composition of transfer functions is written left-to-right.

Like regular expressions.

\[ f_{U,S} = f_1 \]

\[ f_{S,\text{OUT}[2]} = f_2(f_3f_2)^* \]

\[ f_{U,\text{OUT}[4]} = f_1f_4 \]

\[ f_{U,\text{OUT}[2]} = f_1f_2(f_3f_2)^* \]
We can compute the meet of transfer functions: 
\[(f \land g)(x) = f(x) \land g(x).\]
- Needed when we combine transfer functions from header to several latches.

We can compute the closure \(f^*\) for each transfer function \(f\), given by \(f^* = f^0 \land f^1 \land f^2 \land \ldots\)
- Note \(f^0 = \text{identity}, f^1 = f, f^2 = f \text{ composed with } f, \text{ etc.}\)
- Needed when we apply T1 to consume a back edge.
Example: RD’s

- Meet of transfer functions $f(x) = (x - K) \cup G$ and $g(x) = (x - K') \cup G'$ corresponds to the paths of $f$ and $g$ in parallel.
- Thus, $[f \land g](x) = (x - (K \cap K')) \cup (G \cup G')$.
- $f^*(x) = x \cup f(x) \cup f(f(x)) \cup \ldots = x \cup (x - K) \cup G \cup ((x - K \cup G) - K) \cup G \ldots = x \cup G$. 

\[ f(f(x)) \]
Example: Meet

Region U consumes node 5 to make region V.

\[
f_{V,5} = f_{14} \land f_{12}(f_{32})^* \]

\[
f_{U,\text{OUT}[4]} = f_{14} \]

\[
f_{U,\text{OUT}[2]} = f_{12}(f_{32})^* \]
The Region-Based Algorithm

- Start with a pass to compute the transfer functions $f_R$, $f_{R,S}$, and $f_{R,\text{OUT}[B]}$.
  - Work small-to-large from regions of a single block to the whole flow graph.
- Then a pass large-to-small, computing $\text{IN}(B)$ for each block $B$. 
If \( R \) is a region consisting of a single node (block), and \( f \) is the transfer function for that block, then \( f_R = \text{identity} \) and \( f_{R,\text{OUT}[R]} = f \).

- **Note**: we use the same name for the block and the region consisting of only that block.
Apply T2: region R consumes region S to make T.

1. $f_T = f_R$.
2. $f_{T,S} = \text{meet of } f_{R,\text{OUT}[B]} \text{ for all predecessors } B \text{ of the header of } S$.
3. For blocks $B$ in $R$, $f_{T,\text{OUT}[B]} = f_{R,\text{OUT}[B]}$.
4. For blocks $B$ in $S$, $f_{T,\text{OUT}[B]} = f_{T,S} f_{S,\text{OUT}[B]}$. 
### Apply T1: add back edges to region S to make R.

1. Compute $g = \text{meet of } f_{S,\text{OUT}[B]}$ for all predecessors $B$ (a member of S) of the header of S.

2. $f_R = g^*$.
   - Notice that the headers of R and S are the same node, but there are paths within R that are not in S, yet take you to the header of S.

3. For all blocks $B$ in S (and therefore in R) let $f_{R,\text{OUT}[B]} = f_R f_{S,\text{OUT}[B]}$. 
Goal: compute IN(B) (as for the iterative algorithm) for each block B.

Let e be the header of the region R that is the entire flow graph.

Basis: Start with IN(e) = f_R(v_{ENTRY}).
Proceed inward, from larger regions to smaller.

Do nothing when T1 used.

For T2, if region T, with header h’, was constructed from R and S, with R consuming S, and S has header h, then \( \text{IN}(h) = f_{T,S}(\text{IN}(h')) f_S \).
Examples: Top-Down Finale

$IN[1] = f_R(v_{ENTRY}) = f_{*, V, OUT[5]}(v_{ENTRY})$

$IN[5] = f_{V, 5}(IN(1))$
Induction Variables

Affine Mappings
Semilattice for Affine Mappings
Application to Finding Induction Variables

Jeffrey D. Ullman
Stanford University
Symbolic Analysis

- Values $V$ are mappings from variables to expressions involving a selected set of *reference variables*.
- **Example**: $m(a) = 2i$; $m(b) = i + j + 1$.
  - $i$ and $j$ are the reference variables.
- We are going to deal with two complicated types of functions:
  1. **Mappings** from variables to expressions, as above.
  2. **Transfer functions** from mappings (in) to mappings (out).
Reference variables are *basic induction variables* = counts of the number of times around some loop.

- This count is well-defined for natural loops.

Set of values V for the framework = *affine mappings* = each variable is mapped to either:
  1. A linear function of the reference variables, or
  2. NAA = “Not an affine” mapping, or
  3. UNDEF = top element = “nothing known.”
Example: Affine Mapping

\[ m(a) = 2i + 3j + 4 \]
\[ m(b) = \text{NAA} \]
\[ m(c) = 4i + 1 \]
\[ m(d) = \text{UNDEF} \]
Lattice for Each Variable

Note: The lattice for V is the product of one of these for each variable.
Natural application of region-based analysis.
Each loop is a region, and its induction variables can be discovered by a framework based on affine mappings.
Example: A Transfer Function

- Let $f$ be the transfer function associated with a block containing only $a = a + 10$.
- Let $m$ be the input mapping for the block; then $f(m) = m'$, where:
  - $m'(a) = m(a) + 10$.
  - $m'(x) = m(x)$ for all $x \neq a$.
- Book uses the notation $f(m)(a) = m(a) + 10$, etc., to avoid having to name $f(m)$.

Question: What if $m(a)$ is NAA or UNDEF?
The Meet Operator

\[(f \land g)(m)(x) =\]

- \[f(m)(x)\] if \(f(m)(x) = g(m)(x)\).
- \[f(m)(x)\] if \(g(m)(x) = \text{UNDEF}\).
- \[g(m)(x)\] if \(f(m)(x) = \text{UNDEF}\).
- NAA otherwise.
Example: Some Analysis

\[ f_{B_2}(m) \begin{align*} (a) &= m(a) + 5 \\ (b) &= m(b) \end{align*} \]

\[ a = a + 5 \]

\[ f_{B_3}(m) \begin{align*} (a) &= m(a) \\ (b) &= m(a) + 2 \end{align*} \]

\[ b = a + 2 \]

\[ f_{B_4}(m) \begin{align*} (a) &= m(b) + 3 \\ (b) &= m(b) \end{align*} \]

\[ a = b + 3 \]

\[ f_{R,\text{OUT}[B_4]}(m) \begin{align*} (a) &= m(a) + 5 \\ (b) &= m(a) + 2 \end{align*} \]
**Example: Meet**

\[ a = b + 3 \]
\[ b = a + 2 \]
\[ a = a + 5 \]

\[ f_{B_2}(m) \]
\[ (a) = m(a) + 5 \]
\[ (b) = m(b) \]

\[ f_{B_4}(m) \]
\[ (a) = m(a) + 5 \]
\[ (b) = m(a) + 2 \]

\[ f_{S,OUT[B_5]}(m) \]
\[ (a) = m(a) + 5 \]
\[ (b) = \text{NAA (probably)} \]
Handling Loop Regions

- Treat the iteration count $i$ as a basic induction variable.
- If $f(m)(x) = m(x) + c$, then $f^i(m)(x) = m(x) + ci$.
- Some other cases in book.
Example: The Entire Loop

\[ a = a + 5 \]
\[ b = a + 2 \]
\[ f_{S,\text{OUT}[B5]}(m) \]
\[ (a) = m(a) + 5i \]
\[ (b) = \text{NAA} \]
Replace the loop counter variable by one of the induction variables (variables that are mapped to an affine expression of the loop count at the point where the loop count is tested).
Here, the value of $a$ is $m(a)+5i$, where $m(a)$ is the value of $a$ before entering the loop, and $i$ is the number of iterations of the loop so far.

If $i$ is dead on exit from the loop, delete “$i=i+1$” and “$i=0$”.

Since $a=30$ when $i=0$, replace “$i<10?$” by “$a<80?$”.

And since $i$ is an induction variable, if not dead we can just assign $i=10$ at the exit.
Example: Revised Code

- $a = 30$
- $a < 80?$
- $a = a + 5$
- $b = a + 2$
- $a = b + 3$

Flow Diagram:
- From $a = 30$ to $a < 80?$
- From $a < 80?$ to $a = a + 5$
- From $a = a + 5$ to $b = a + 2$
- From $b = a + 2$ to $a = b + 3$
- From $a = b + 3$ to $B_5$