Topic-I-C

Dataflow Analysis
Global Dataflow Analysis

Motivation

We need to know variable **def** and **use** information between basic blocks for:

- constant folding
- dead-code elimination
- redundant computation elimination
- code motion
- induction variable elimination
- build data dependence graph (DDG)
- etc.
Topics of DataFlow Analysis

- Reaching definition
- Live variable analysis
- ud-chains and du-chains
- Available expressions
- Others ..
1. Definition & Use

\[ S: v_1 = \ldots v_2 \]

\( S \) is a “definition” of \( v_1 \)

\( S \) is a “use” of \( v_2 \)
Compute Def and Use Information of a Program P?

- Case 1: P is a basic block?
- Case 2: P contains more than one basic blocks?
Points and Paths

In the example, how many points basic block B1, B2, B3, and B5 have?

B1 has four, B2, B3, and B5 have two points each.

**points** in a basic block:
- between statements
- before the first statement
- after the last statement

In the diagram:
- **d₁**: i := m - 1
- **d₂**: j := n
- **d₃**: a := u₁
- **d₄**: i := i + 1
- **d₅**: j := j + 1
- **d₆**: a := u₂
Points and Paths

A path is a sequence of points \( p_1, p_2, \ldots, p_n \) such that either:
(i) if \( p_i \) immediately precedes \( S \), then \( p_{i+1} \) immediately follows \( S \).
(ii) or \( p_i \) is the end of a basic block and \( p_{i+1} \) is the beginning of a successor block.

In the example, is there a path from the beginning of block B5 to the beginning of block B6?

Yes, it travels through the end point of B5 and then through all the points in B2, B3, and B4.
Reach and Kill

**Kill**

A definition $d_1$ of a variable $v$ is killed between $p_1$ and $p_2$ if in every path from $p_1$ to $p_2$ there is another definition of $v$.

**Reach**

A definition $d$ reaches a point $p$ if $\exists$ a path $d \rightarrow p$, and $d$ is not killed along the path.

In the example, do $d_1$ and $d_2$ reach the points and $\bullet$?
Reach Example

The set of defs reaching the use of $N$ in S8: \{S2, S8\}
def S2 reach S11 along statement path: (S2, S3, S4, S5, S6, S7, S11)
S8 reach S11 along statement path: (S8, S9, S10, S5, S6, S7, S11)
Problem Formulation:
Example 1

Can $d_1$ reach point $p_1$?

$d_1$  $x := \exp \text{p}$

$s_1$  if $p > 0$

$s_2$  $x := x + 1$

$s_3$  $a = b + c$

$s_4$  $e = x + 1$

It depends on what point $p_1$ represents!!!
Problem Formulation: Example 2

Can $d_1$ and $d_4$ reach point $p_3$?

d_1 \quad x := \text{exp1}

s_2 \quad \text{while } y > 0 \text{ do}

s_3 \quad a := b + 2

s_4 \quad x := \text{exp2}

s_5 \quad c := a + 1

end while

p_3

$\text{Problem Formulation: Example 2}$

Can $d_1$ and $d_4$ reach point $p_3$?

$d_1 \quad x := \text{exp1}$

$s_2 \quad \text{while } y > 0 \text{ do}$

$s_3 \quad a := b + 2$

$p_3$

$s_4 \quad x := \text{exp2}$

$s_5 \quad c := a + 1$

end while
Data-Flow Analysis of Structured Programs

Structured programs have an useful property: there is a single point of entrance and a single exit point for each statement.

We will consider program statements that can be described by the following syntax:

Statement → \text{id} := \text{Expression} \\
| \text{Statement ; Statement} \\
| \text{if} \ \text{Expression} \ \text{then} \ \text{Statement} \ \text{else} \ \text{Statement} \\
| \text{do} \ \text{Statement} \ \text{while} \ \text{Expression} \\

Expression → \text{id} + \text{id} \\
| \text{id}
Structured Programs

This restricted syntax results in the forms depicted below for flowgraphs

S ::= id := E
| S ; S
| if E then S else S
| do S while E

E ::= id + id
| id

This restricted syntax results in the forms depicted below for flowgraphs
Data-Flow Values

1. Each program point associates with a data-flow value
2. A data-flow value represents the possible program states that can be observed for that program point.
3. The data-flow value depends on the goal of the analysis.

Given a statement $S$, $\text{in}(S)$ and $\text{out}(S)$ denote the data-flow values before and after $S$, respectively.
Assume basic block $B$ consists of statement $s_1, s_2, ..., s_n$ ($s_1$ is the first statement of $B$ and $s_n$ is the last statement of $B$), the data-flow values immediately before and after $B$ is denoted as:

$$in(B) = in(s_1)$$
$$out(B) = out(s_n)$$
Instances of Data-Flow Problems

- Reaching Definitions
- Live-Variable Analysis
- DU Chains and UD Chains
- Available Expressions

To solve these problems we must take into consideration the data-flow and the control flow in the program. A common method to solve such problems is to create a set of data-flow equations.
Iterative Method for Dataflow Analysis

- Establish a set of dataflow relations for each basic block
- Establish a set dataflow equations between basic blocks
- Establish an initial solution
- Iteratively solve the dataflow equations, until a fixed point is reached.
Generate set: \( \text{gen}(S) \)

In general, \( d \in \text{gen}(S) \) if \( d \) reaches the end of \( S \) independent of whether it reaches the beginning of \( S \).

We restrict \( \text{gen}(S) \) contains only the definition in \( S \).

If \( S \) is a basic block, \( \text{gen}(S) \) contains all the definitions inside the basic block that are “visible” immediately after the block.
\[ d \in \text{kill}(S) \Rightarrow d \text{ never reaches the end of } S. \]

This is equivalent to say:
\[ d \text{ reaches end of } S \Rightarrow d \notin \text{kill}(S) \]

Of course the statements \( d_1, d_2, \ldots, d_k \) all get killed except \( dd \) itself.

A basic block’s kill set is simply the union of all the definitions killed by its individual statements!
Reaching Definitions

Problem Statement:

Given a program and a program point determine the set of definitions reaching this point in a program.
Iterative Algorithm for Reaching Definitions

**dataflow equations**

The set of definitions reaching the entry of basic block $B$:

$$in(B) = \bigcup_{P \in \text{predecessor}(B)} out(P)$$

The set of definitions reaching the exit of basic block $B$:

$$out(B) = gen(B) \cup \{ in(B) - \text{kill}(B) \}$$
Iterative Algorithm for Reaching Definitions

**Algorithm**

1) $\text{out}(ENTRY) = \emptyset$;

2) for (each basic block $B$ other than $ENTRY$)
   
   $\text{out}(B) = \emptyset$;

3) while (changes to any out occur)

4) for (each $B$ other than $ENTRY$)
   
   $\text{in}(B) = \bigcup_{P \in \text{predecessors of } B} \text{out}(P)$;

   $\text{out}(B) = \text{gen}(B) \cup (\text{in}(B) - \text{kill}(b))$;

Need a flag to test if a out is changed! The initial value of the flag is true.
Dataflow Equations – a simple case

\[ \text{gen}(S) = \{d\} \]
\[ \text{kill}(S) = D_a - \{d\} \]

\[ \text{gen}(S) = \text{gen}(S_2) \cup (\text{gen}(S_1) - \text{kill}(S_2)) \]
\[ \text{kill}(S) = \text{kill}(S_2) \cup (\text{kill}(S_1) - \text{gen}(S_2)) \]

\[ \text{gen}(S) = \text{gen}(S_1) \cup \text{gen}(S_2) \]
\[ \text{kill}(S) = \text{kill}(S_1) \cap \text{kill}(S_2) \]

\[ \text{gen}(S) = \text{gen}(S_1) \]
\[ \text{kill}(S) = \text{kill}(S_1) \]

Da is the set of all definitions in the program for variable a!
Dataflow Equations

\[
\text{out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S))
\]

\[
\text{in}(S) = \text{in}(S_1)
\]
\[
\text{in}(S_2) = \text{out}(S_1)
\]
\[
\text{out}(S) = \text{out}(S_2)
\]

\[
\text{in}(S) = \text{in}(S_1) = \text{in}(S_2)
\]
\[
\text{out}(S) = \text{out}(S_1) \cup \text{out}(S_2)
\]

\[
\text{in}(S) = \text{in}(S_1) \cup \text{out}(S_1)
\]
\[
\text{out}(S) = \text{out}(S_1)
\]

Date-flow equations for reaching definitions
Dataflow Analysis: An Example

Using RD (reaching def) as an example:

\[ \text{in}(L) = \{d_1\} \cup \text{out}(L) \]
\[ \text{gen}(L) = \{d_2\} \]
\[ \text{kill}(L) = \{d_1\} \]
\[ \text{out}(L) = \text{gen}(L) \cup \{\text{in}(L) - \text{kill}(L)\} \]

\( \text{in}(L) \) depends on \( \text{out}(L) \), and \( \text{out}(L) \) depends on \( \text{in}(L) \)!!
Initialization

\[ \text{out}[L] = \emptyset \]

First iteration

\[ \text{out}(L) = \text{gen}(L) \cup (\text{in}(L) - \text{kill}(L)) \]
\[ = \{d_2\} \cup (\{d_1\} - \{d_1\}) \]
\[ = \{d_2\} \]

Second iteration

\[ \text{out}(L) = \text{gen}(L) \cup (\text{in}(L) - \text{kill}(L)) \]

but now:

\[ \text{in}(L) = \{d_1\} \cup \text{out}(L) = \{d_1\} \cup \{d_2\} \]
\[ = \{d_1, d_2\} \]

therefore:

\[ \text{out}(L) = \{d_2\} \cup (\{d_1, d_2\} - \{d_1\}) \]
\[ = \{d_2\} \cup \{d_2\} \]
\[ = \{d_2\} \]

So, we reached the fixed point!

\[ \text{in}(L) = \{d_1\} \cup \text{out}(L) \]
\[ \text{gen}(L) = \{d_2\} \]
\[ \text{kill}(L) = \{d_1\} \]
\[ \text{out}(L) = \text{gen}(L) \cup \{\text{in}(L) - \text{kill}(L)\} \]
Reaching Definitions: Another Example

Step 1: Compute gen and kill for each basic block

\[
\begin{align*}
\text{gen}(B_1) &= \{d_1, d_2, d_3\} \\
\text{kill}(B_1) &= \{d_4, d_5, d_6, d_7\}
\end{align*}
\]

\[
\begin{align*}
\text{gen}(B_2) &= \{d_4, d_5\} \\
\text{kill}(B_2) &= \{d_1, d_2, d_7\}
\end{align*}
\]

\[
\begin{align*}
\text{gen}(B_3) &= \{d_6\} \\
\text{kill}(B_3) &= \{d_3\}
\end{align*}
\]

\[
\begin{align*}
\text{gen}(B_4) &= \{d_7\} \\
\text{kill}(B_4) &= \{d_1, d_4\}
\end{align*}
\]
Step 2: For every basic block, make:
\[ \text{out}[B] = \emptyset \]

**Initialization:**

\[ \text{out}(B_1) = \emptyset \]
\[ \text{out}(B_2) = \emptyset \]
\[ \text{out}(B_3) = \emptyset \]
\[ \text{out}(B_4) = \emptyset \]
Reaching Definitions: Another Example (Con’t)

To simplify the representation, the in[B] and out[B] sets are represented by bit strings. Assuming the representation $d_1 d_2 d_3 d_4 d_5 d_6 d_7$ we obtain:

**Initialization:**

- $\text{out}(B_1) = \emptyset$
- $\text{out}(B_2) = \emptyset$
- $\text{out}(B_3) = \emptyset$
- $\text{out}(B_4) = \emptyset$

<table>
<thead>
<tr>
<th>Block</th>
<th>in[B]</th>
<th>out[B]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_2</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_3</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_4</td>
<td>000 0000</td>
<td></td>
</tr>
</tbody>
</table>

Notation: $d_1 d_2 d_3 d_4 d_5 d_6 d_7$
While a fixed point is not found:

\[ \text{in}(B) = \bigcup_{P \in \text{pred}(B)} \text{out}(P) \]

\[ \text{out}(B) = \text{gen}(B) \cup (\text{in}(B) - \text{kill}(B)) \]

**ENTRY**

- **B1**
  - \( d_1: i := m - 1 \)
  - \( d_2: j := n \)
  - \( d_3: a := u_1 \)

- **B2**
  - \( d_4: i := i + 1 \)
  - \( d_5: j := j - 1 \)

- **B3**
  - \( d_6: a := u_2 \)

- **B4**
  - \( d_7: i := u_3 \)

**EXIT**

Notation: \( d_1d_2d_3d_4d_5d_6d_7 \)

**Notation:**
- \( \text{gen}(B_2) = \{d_1, d_2, d_3\} \)
- \( \text{kill}(B_1) = \{d_4, d_5, d_6, d_7\} \)
- \( \text{gen}(B_2) = \{d_4, d_5\} \)
- \( \text{kill}(B_2) = \{d_1, d_2, d_7\} \)
- \( \text{gen}(B_3) = \{d_6\} \)
- \( \text{kill}(B_3) = \{d_3\} \)
- \( \text{gen}(B_4) = \{d_7\} \)
- \( \text{kill}(B_4) = \{d_1, d_4\} \)

**Table:**

<table>
<thead>
<tr>
<th>Block</th>
<th>Initial</th>
<th>First Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[B]</td>
<td>out[B]</td>
</tr>
<tr>
<td>B1</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B2</td>
<td>000 0000</td>
<td>000 1100</td>
</tr>
<tr>
<td>B3</td>
<td>000 0000</td>
<td>000 0110</td>
</tr>
<tr>
<td>B4</td>
<td>000 0000</td>
<td>000 0001</td>
</tr>
</tbody>
</table>

**Initial Blocks:**
- \( B_1 \)
  - 000 0000
- \( B_2 \)
  - 000 0000
- \( B_3 \)
  - 000 0000
- \( B_4 \)
  - 000 0000

**First Iteration:**
- \( B_1 \) in[B] = 000 0000, out[B] = 111 0000
- \( B_2 \) in[B] = 000 0000, out[B] = 000 1100
- \( B_3 \) in[B] = 000 0000, out[B] = 000 0110
- \( B_4 \) in[B] = 000 0000, out[B] = 000 0000

**Out(B) = gen(B)**
while a fixed point is not found:

\[
\text{in}(B) = \bigcup_{P \in \text{pred}(B)} \text{out}(P)
\]
\[
\text{out}(B) = \text{gen}(B) \cup (\text{in}(B) - \text{kill}(B))
\]

Notation: \(d_1d_2d_3d_4d_5d_6d_7\)
Reaching Definitions: Another Example (Con’t)

while a fixed point is not found:

\[
in[B] = \bigcup \text{out}[P] \quad \text{where } P \text{ is a predecessor of } B
\]

\[
\text{out}[B] = \text{gen}[B] \cup (\text{in}[B]-\text{kill}[B])
\]

<table>
<thead>
<tr>
<th>Block</th>
<th>Second Iteration</th>
<th>Third Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B2</td>
<td>111 0010</td>
<td>001 1110</td>
</tr>
<tr>
<td>B3</td>
<td>000 1100</td>
<td>000 1100</td>
</tr>
<tr>
<td>B4</td>
<td>000 1100</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

we reached the fixed point!
while a fixed point is not found: 
\[ \text{in}[B] = \bigcup \text{out}[P] \] 
where P is a predecessor of B
\[ \text{out}[B] = \text{gen}[B] \cup (\text{in}[B]-\text{kill}[B]) \]

Notation: \[ d_1d_2d_3d_4d_5d_6d_7 \]

we reached the fixed point!
Other Applications of Data flow Analysis

- Live Variable Analysis
- DU and UD Chains
- Available Expressions
- Constant Propagation
- Constant Folding
- Others ..
A variable $V$ is live at the exit of a basic block $n$, if there is a def-free path from $n$ to an outward exposed use of $V$ in a node $n'$.

“Live variable analysis problem” – determine the set of variables which are live at the exit from each program point.

Live variable analysis is a “backwards dataflow” analysis, that is the analysis is done in a backwards order.
Live Variable Analysis: Another Example of Flow Analysis

L1: b := 3;
L2: c := 5;
L3: a := b + c;
goto L1;

The set of live variables at line L2 is \{b, c\}, but the set of live variables at line L1 is only \{b\} since variable "c" is updated in line 2. The value of variable "a" is never used, so the variable is never live.

Copy from Wikipedia, the free encyclopedia
Live Variable Analysis: Def and use set

- \textit{def}(B): the set of variables defined in basic block \( B \) prior to any use of that variable in \( B \)
- \textit{use}(B): the set of variables whose values may be used in \( B \) prior to any definition of the variable.
Live Variable Analysis

**Dataflow Equations**

The set of variables live at the entry of basic block $B$:

$$in(B) = use(B) \cup \{out(B) - def(B)\}$$

The set of variables live at the exit of basic block $B$:

$$out(B) = \bigcup_{S \in \text{successors}(B)} in(S)$$
Iterative Algorithm for Live Variable Analysis

Algorithm

1) out(EXIT) = ∅;
2) for (each basic block B other than EXIT) in(B) = ∅;
3) while (changes to any “in” occur)
4) for (each B other than EXIT)
   
   \[
   \text{out}(B) = \bigcup_{S \in \text{successors}(B)} \text{in}(S)
   \]

   \[
   \text{in}(B) = \text{use}(B) \cup \{\text{out}(B) - \text{def}(B)\}
   \]
Live Variable Analysis: a Quiz

Calculate the live variable sets $\text{in}(B)$ and $\text{out}(B)$ for the program:

1. $d_1$: $i := m - 1$
2. $d_2$: $j := n$
3. $d_3$: $a := u_1$
4. $d_4$: $i := i + 1$
5. $d_5$: $j := j - 1$
6. $d_6$: $a := u_2$
7. $d_7$: $i := u_3$

Diagram:

- **B1**: $d_1$, $d_2$, $d_3$
- **B2**: $d_4$, $d_5$
- **B3**: $d_6$
- **B4**: $d_7$
Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression.

Def-Use (D-U), and Use-Def (U-D) chains are efficient data structures that keep this information.

Notice that when a code is represented in Static Single-Assignment (SSA) form (as in most modern compilers) there is no need to maintain D-U and U-D chains.
UD Chain

An UD chain is a list of all definitions that can reach a given use of a variable.

A UD chain: $UD(S_n, v) = (S_1', ..., S_m')$. 

\[ \begin{align*} 
S_n: \ldots = \ldots \ v \ldots \\
S_1': v = \ldots \\
S_m': v = \ldots
\end{align*} \]
A DU chain is a list of all uses that can be reached by a given definition of a variable. DU Chain is a counterpart of a UD Chain.

**DU Chain**

\[
DU(S'_n, v) = (S_1, \ldots, S_k).
\]

A DU chain: \(DU(S'_n, v) = (S_1, \ldots, S_k)\).
Use of DU/UD Chains in Optimization/Parallelization

- Dependence analysis
- Live variable analysis
- Alias analysis
- Analysis for various transformations
Available Expressions

An expression $x + y$ is available at a point $p$ if:

1. Every path from the start node to $p$ evaluates $x + y$.

2. After the last evaluation prior to reaching $p$, there are no subsequent assignments to $x$ or $y$.

We say that a basic block kills expression $x + y$ if it may assign $x$ or $y$, and does not subsequently recompute $x + y$. 
Available Expression: Example

Yes. It is generated in all paths leading to B4 and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expression: Example

S1: X = A * B
S2: Z = X + C

B1

S3: Y = A * B
S4: W = Y + C

B2

S5: C = 1

B3

S6: T = A * B
S7: V = D * T

B4

Yes. It is generated in all paths leading to B4 and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expression: Example

Available Expression:

Example

S1: \( \text{temp} = A \times B \)
S2: \( Z = \text{temp} + C \)

S3: \( \text{temp} = A \times B \)
S4: \( W = \text{temp} + C \)

S5: \( C = 1 \)

S6: \( T = \text{temp} \)
S7: \( V = D \times T \)

Yes. It is generated in all paths leading to \( B_4 \) and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expressions: gen and kill set

Assume $U$ is the “universal” set of all expressions appearing on the right of one or more statements in a program.

\[ e_{gen}(B) : \text{the set of expressions generated by } B \]

\[ e_{kill}(B) : \text{the set of expressions in } U \text{ killed in } B. \]
Calculate the Generate Set of Available Expressions

No generated expression

\[ x = y + z \]

\[ p \]

\[ S: \emptyset \]

\[ q \]

\[ S': \text{add } y+z \text{ to } S; \text{ delete expressions involving } x \text{ from } S \]

\[ S \]

\[ \emptyset \]

\[ a = b + c \]

\[ b = a - d \]

\[ c = b + c \]

\[ d = a - d \]

\[ b + c \]

\[ a - d, b + c \]

\[ a - d \]
Iterative Algorithm for Available Expressions

**dataflow equations**

The set of expressions available at the entry of basic block $B$:

$$in(B) = \bigcap_{P \in \text{predecessors}(B)} out(P)$$

The set of expressions available at the exit of basic block $B$:

$$out(B) = e_{gen}(B) \cup \{ in(B) - e_{kill}(B) \}$$
Iterative Algorithm for Reaching Definitions

Algorithm

1. \( \text{out}(\text{ENTRY}) = \emptyset; \)
2. For each basic block \( B \) other than \( \text{ENTRY} \)
   \( \text{out}(B) = U \)
3. While changes to any \( \text{out} \) occur
4. for each \( B \) other than \( \text{ENTRY} \)
   \[ \{ \text{in}(B) = \bigcap_{P \in \text{predecessors}(B)} \text{out}(P) \} \]
   \[ \text{out}(B) = e_{\text{gen}}(B) \cup \{ \text{in}(B) - e_{\text{kill}}(B) \} \]
Use of Available Expressions

- Detecting global common subexpressions
Constant propagation is the process of substituting the values of known constants in expressions at compile time.

Constant folding is a compiler optimization technique where constant sub-expressions are evaluated at compiler time.
i = 32*48-1530 → i = 6

Constant folding can be implemented:

• In a compiler’s front end on the IR tree (before it is translated into three-address codes)

• In the back end, as an adjunct to constant propagation
Constant Propagation Example

```java
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

**Constant propagation**

```java
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

**Constant folding**

```java
int x = 14;
int y = 0;
return 0;
```
Summary

- Basic Blocks
- Control Flow Graph (CFG)
- Dominator and Dominator Tree
- Natural Loops
- Program point and path
- Dataflow equations and the iterative method
- Reaching definition
- Live variable analysis
- Available expressions
Remarks of Mathematical Foundations on Solving Dataflow Equations

• As long as the dataflow value domain is “nice” (e.g. semi-lattice)
• And each function specified by the dataflow equation is “nice” -- then iterative application of the dataflow equations at each node will eventually terminate with a stable solution (a fix point).
• For mathematical foundation -- read
  • Muchnik’s book: Section 8.2, pp 223

For a good discussion: also read 9.3 (pp 618-632) in new Dragon Book
Algorithm Convergence

Intuitively we can observe that the algorithm converges to a fix point because the $\text{out}(B)$ set never decreases in size.

It can be shown that an upper bound on the number of iterations required to reach a fix point is the number of nodes in the flow graph.

Intuitively, if a definition reaches a point, it can only reach the point through a cycle free path, and no cycle free path can be longer than the number of nodes in the graph.

Empirical evidence suggests that for real programs the number of iterations required to reach a fix point is less than five.
More remarks

If a data-flow framework meets “good” conditions then it has a unique fixed-point solution

The iterative algorithm finds the (best) answer

The solution does not depend on order of computation

Algorithm can choose an order that converges quickly
Ordering the Nodes to Maximize Propagation

- Reverse postorder visits predecessors before visiting a node
- Use reverse preorder for backward problems
  - Reverse postorder on reverse CFG is reverse preorder
Iterative solution to general data-flow frameworks

**INPUT:** A data-flow framework with the following components:

1. A data-flow graph, with specially labeled ENTRY and EXIT nodes,
2. A direction of the data-flow $D$,
3. A set of values $V$,
4. A meet operator $\Lambda$,
5. A set of functions $F$, where $f_B$ in $F$ is the transfer function for block $B$, and
6. A constant value $v_{\text{ENTRY}}$ or $v_{\text{EXIT}}$ in $V$, representing the boundary condition for forward and backward frameworks, respectively.

**OUTPUT:** Values in $V$ for $\text{IN}[B]$ and $\text{OUT}[B]$ for each block $B$ in the data-flow graph.

*(From p925 in Dragon Book Edition 2)*
Iterative algorithm for a forward data-flow problem

1. \( \text{OUT}(\text{ENTRY}) = V_{\text{ENTRY}}; \)
2. \( \text{for} \) (each basic block \( B \) other than \( \text{ENTRY} \)) \( \text{OUT}(B) = T; \)
3. \( \text{while} \) (changes to any \( \text{OUT} \) occur)
4. \( \text{for} \) (each basic block \( B \) other than \( \text{ENTRY} \)) \{ 
   \[
   \text{IN}(B) = \bigcap_{P \in \text{predecessors}(B)} \text{OUT}(P);
   \]
   \[
   \text{OUT}(B) = f_B(\text{IN}(B))
   \]
\}

(From p926 in Dragon Book Edition 2)
Iterative algorithm for a backward data-flow problem

1. \( \text{IN(EXIT)} = V_{EXIT}; \)
2. \textit{for (each basic block B other than EXIT)} \( \text{IN(B)} = T; \)
3. \textit{while (changes to any IN occur)}
4. \textit{for (each basic block B other than EXIT)} \{ \begin{align*}
\text{OUT(B)} & = \bigcap_{S \in \text{successors(B)}} \text{IN(S)} \\
\text{IN(B)} & = f_B(\text{OUT(B)}); \\
\end{align*} \}

(From p926 in Dragon Book Edition 2)