Data-Flow Analysis

D Dominators to Reaching Definitions

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Citation numbers refer to entries in the EaC2e bibliography.
Data-flow Analysis

**Definition**

*Data-flow analysis (DFA) is a collection of techniques for compile-time reasoning about the run-time flow of values*

- We use the results of DFA to prove safety & identify opportunities
  - Not an end unto itself

- Almost always involves building a graph
  - Control-flow graph, call graph, or graphs derived from them
  - Sparse evaluation graphs to model flow of values *(efficiency)*

- Usually formulated as a set of *simultaneous equations*
  - Sets attached to nodes and edges
  - Often use sets with a lattice or semilattice structure

We have seen several data-flow problems.
Prior Examples

Computing LIVEOUT Sets

• Domain is the set of variable names in the procedure
• Data-flow equations define LIVE at the end of a block, LIVEOUT

**Initialization:**

\[
\text{LIVEOUT}(n) = \emptyset, \forall n
\]

**Fixed-point equations:**

\[
\text{LIVEOUT}(b) = \bigcup_{s \in \text{succs}(b)} (\text{UEVAR}(b) \cup (\text{LIVEOUT}(b) \cap \text{VARKILL}(b)))
\]

LIVE is a backward-flow problem

where

- \(\text{UEVAR}(b)\) is the set of names used in \(b\) before definition in \(b\)
- \(\text{VARKILL}(b)\) is the set of names defined in \(b\)
Prior Examples

Computing AVAIL Sets

• Domain is the set of expressions computed in the procedure
• Data-flow equations are more complex

**Initialization:**

\[
\begin{align*}
\text{AVAIL}(n_0) &= \emptyset \\
\text{AVAIL}(n) &= \emptyset, \quad \forall \ n \neq n_0
\end{align*}
\]

**Fixed-point equations:**

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) \cap \text{EXPRKILL}(b)))
\]

where

- \( \text{DEEPXR}(b) \) is the set of expressions defined in \( b \) and not subsequently killed in \( b \)
- \( \text{EXPRKILL}(b) \) is the set of expressions killed in \( b \) because one or more operand is redefined in \( b \)

AVAIL is a forward-flow problem
Prior Examples

Global constant propagation

B1: $\{b=3, c=4\}$  B2: $\{b=1, c=6\}$

B3: $a \leftarrow b + c$

Function “$f_3$” models the effect of block B3

- $f_3(\{b=3, c=4\}) \Rightarrow \{a=7\}$
- $f_3(\{b=1, c=6\}) \Rightarrow \{a=7\}$
- $f_3(B1 \land B2) = f_3(\emptyset) \Rightarrow \{a=\bot\}$

Result depends on order of evaluation of the $\land$ operation and application of $f$

- First condition in admissibility
  $\forall f \in F, \forall x, y \in L, f(x \land y) = f(x) \land f(y)$
- Constant propagation is not admissible
  - Kam & Ullman time bound does not hold
  - There are tight time bounds, however, based on lattice height
  - Require a variable-by-variable formulation ...

Because meet does not distribute over function application, constant propagation is not “admissible” in the Kam-Ullman sense.
Prior Examples

Interprocedural May Modify sets

\[
\text{shift}(a, b, c, d, e, f)
\]

\{
    \text{local} \ t;
    \text{...}
    \text{call} \ \text{shift}(t, a, b, c, d, e);
    f = 1;
    \text{...}
\}

- Iterations proportional to number of parameters
  - Not a function of the call graph
  - Can make example arbitrarily bad
- Proportional to length of chain of bindings...

- Assume call-by-reference
- Compute the set of variables (in namespace of \text{shift}) that can be modified by a call to \text{shift}
- How long does it take?

Nothing to do with \(d(G)\)

Call-by-reference parameters plus recursion make the summary problems fail the Kam-Ullman “rapid” condition.

(COMP 512, Rice University)
Proliferation of GDFAPs

In the late 1960’s and the 1970’s many data-flow problems were proposed

• GDFAP became the standard way to prove safety of a transformation
  ♦ New transformation implied new GDFAP
  ♦ Optimizing compilers spent a large fraction of compile time solving GDFAPs
  ♦ Computers were relatively slow (1 – 10 MIPS) and small (16 to 32 MB)
  ♦ Development of “frameworks” for GDFA

• Many papers showed a new GDFAP & a new transformation
  ♦ Other applications arose for the GDFAP technology
  ♦ See the papers on “DAVE” by Osterweil et al.
More GDFAPS: Very Busy Expressions

An expression $e$ is *very busy* on exit from block $b$, *iff* $e$ is evaluated & used along every path from $b$ to $n_f$ and evaluating $e$ at the end of $b$ would produce the same result as the next evaluation along those paths.

The Plan

- Annotate each block $n$ with a set $\text{VERYBUSY}(b)$ that contains expressions
  - Solve data-flow equations (standard iterative solver)
- If $e$ is in $\text{VERYBUSY}(b)$, insert an evaluation at the end of $n$ and eliminate the subsequent evaluations that it covers
  - If $e$ is in $\text{VERYBUSY}(b)$ for successive blocks, want to insert it in the “right” block
  - Might be the last $b$ (minimize register demand) or least frequently executed $b$ (minimize dynamic number of evaluations) or …
- This optimization aims, primarily, to reduce code space
More GDFAPS: Very Busy Expressions

Transformation: Hoisting

• \( e \) defined in every successor of \( b \)
• Evaluating \( e \) in \( b \) produces same result
• Saves code space, but shortens no path

Data-flow problem: Very Busy Expressions

\[
\text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} (\text{UEEXPR}(s) \cup (\text{VERYBUSY}(s) \cap \text{EXPRKILL}(s)))
\]

\[
\text{VERYBUSY}(n_f) = \emptyset
\]

• \( \text{VERYBUSY}(b) \) contains expressions that are very busy at end of \( b \)
• \( \text{UEEXPR}(b) \) is the set of expressions used before they are killed in \( b \)
• \( \text{EXPRKILL}(b) \) is the set of expressions killed before they are used in \( b \)

\( \text{VERYBUSY} \) expressions is a **backward** flow problem

\(|\text{VERYBUSY}| = |\text{expressions}|\)
More GDFAPS: Constant Propagation (Classic formulation)

Transformation: Global Constant Folding
- Along every path to \( p \), \( v \) has same known value
- Specialize computation at \( p \) based on \( v \)'s value

Data-flow problem: Constant Propagation
Domain is the set of pairs \(<v_i,c_i>\) where \( v_i \) is a variable and \( c_i \in C \)

\[
\text{CONSTANTS}(b) = \bigwedge_{p \in \text{preds}(b)} f_p(\text{CONSTANTS}(p))
\]
- \( \bigwedge \) performs a pairwise meet on two sets of pairs
- \( f_p(x) \) is a block specific function that models the effects of block \( p \) on the \(<v_i,c_i>\) pairs in \( x \)

Constant propagation is a **forward** flow problem

\[|\text{CONSTANTS}| = |\text{variables}|\]
More GDFAPs: Constant Folding

Meet operation is more complex than we have already seen

• $c_1 \land c_2 = c_1$ if $c_1 = c_2$, else ⊥ (bottom & top as expected)

What about $f_p$?

• If $p$ has one statement then
  $x \leftarrow y$ with $\text{CONSTANTS}(p) = \{ ...<x,l_1>, ...<y,l_2>,...\}$
  then $f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2>$

• If $p$ has $n$ statements then
  $f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(...f_1(\text{CONSTANTS}(p))...)))$
  where $f_i$ is the function generated by the $i^{th}$ statement in $p$

• $c_1 \land c_2 = c_1$ if $c_1 = c_2$, else ⊥

$\forall c_1 \land c_2 = c_1$ if $c_1 = c_2$, else ⊥ (bottom & top as expected)

What about $f_p$?

• If $p$ has one statement then
  $x \leftarrow y$ with $\text{CONSTANTS}(p) = \{ ...<x,l_1>, ...<y,l_2>,...\}$
  then $f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2>$

• If $p$ has $n$ statements then
  $f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(...f_1(\text{CONSTANTS}(p))...)))$
  where $f_i$ is the function generated by the $i^{th}$ statement in $p$

Constant propagation, in its more general forms, can become intractable because it encodes arithmetic.
Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j − 1

If target, taken, or not_taken are ambiguous, then we must include all labeled ops as leaders.

Sources of ambiguous targets:
- Fall-through branch path
- Jump to a register

No Ambiguity In ILOC:
All branches in ILOC have two explicit targets. Branches and jumps target a label rather than a register.
In the original compiler, jump to register was followed with an advisory list of labels generated when the ILOC was generated.
Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

// find all the leaders, assume first op & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)
// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j – 1

EXAMPLE

0 a ← 4
1 t1 ← a * 4
2 L1: t2 ← t1 / c
3 if t2 < w then goto L2
4 m ← t1 * k
5 t3 ← m + i
6 L2: h ← i
7 m ← t3 – h
8 if t3 ≥ 0 then goto L3
9 goto L1
10 L3: halt
Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
  if op[i] is a jump
    then leader[next++] ← target(i)
  if op[i] is a branch then
    leader[next++] ← taken(i)
    leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next – 1
  j ← leader[i] + 1
  while j ≤ n and j ∉ leader
    j ← j + 1
  last[i] ← j – 1

EXAMPLE

0  a ← 4
1  t1 ← a * 4
2  L1:  t2 ← t1 / c
3  if t2 < w then goto L2
4  m ← t1 * k
5  t3 ← m + i
6  L2:  h ← i
7  m ← t3 – h
8  if t3 ≥ 0 then goto L3
9  goto L1
10 L3:  halt

<table>
<thead>
<tr>
<th>LEADER</th>
<th>0</th>
<th>6</th>
<th>4</th>
<th>9</th>
<th>10</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

// find all the leaders, assume first op
// & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
        then leader[next++] ← target(i)
    if op[i] is a branch then
        leader[next++] ← taken(i)
        leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next – 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j – 1

EXAMPLE

0 a ← 4
1 t1 ← a * 4
2 L1: t2 ← t1 / c
3 if t2 < w then goto L2
4 m ← t1 * k
5 t3 ← m + i
6 L2: h ← i
7 m ← t3 – h
8 if t3 ≥ 0 then goto L3
9 goto L1
10 L3: halt

LEADER

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LAST

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Building a Control-Flow Graph

The first step in almost any data-flow analysis is building a CFG

// find all the leaders, assume first op // & block are numbered zero
next ← 0
leader[next] ← 0
for i ← 0 to n
    if op[i] is a jump
       then leader[next++] ← target(i)
    if op[i] is a branch then
       leader[next++] ← taken(i)
       leader[next++] ← not_taken(i)

// build all the blocks
for i ← 0 to next − 1
    j ← leader[i] + 1
    while j ≤ n and j ∉ leader
        j ← j + 1
    last[i] ← j − 1

EXAMPLE

0 a ← 4
1 t1 ← a * 4
2 L1: t2 ← t1 / c
3 if t2 < w then goto L2
4 m ← t1 * k
5 t3 ← m + i
6 L2: h ← i
7 m ← t3 − h
8 if t3 ≥ 0 then goto L3
9 goto L1
10 L3: halt

LEADER 0 6 4 9 10 2
LAST 1 8 5 9 10 3
Dominators

Definitions

In a flow graph, \( x \) dominates \( y \) if and only if every path from the entry of the control-flow graph to the node for \( y \) includes \( x \)

- By definition, \( x \) dominates \( x \)
- We associate a DOM set with each node
- \( |\text{DOM}(x)| \geq 1 \)

Immediate dominator

- For any node \( x \), there must be a \( y \) in \( \text{DOM}(x) \) closest to \( x \)
  - Unless \( x = n_0 \), \( x \neq \text{IDOM}(x) \)
- We call this \( y \) the immediate dominator of \( x \)
- As a matter of notation, we write this as \( \text{IDOM}(x) \)

Dominators

Dominators have many uses in analysis & transformation

- Finding loops
- Building SSA form
- Making code motion decisions

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>

Dominator tree:
- Node A
  - m₀ ← a + b
  - n₀ ← a + b
- Node B
  - p₀ ← c + d
  - r₀ ← c + d
- Node C
  - q₀ ← a + b
  - r₁ ← c + d
- Node D
  - e₀ ← b + 18
  - s₀ ← a + b
  - u₀ ← e + f
- Node E
  - e₁ ← a + 17
  - t₀ ← c + d
  - u₁ ← e + f
- Node F
  - e₂ ← φ(e₀,e₁)
  - u₂ ← φ(u₀,u₁)
  - v₀ ← a + b
  - w₀ ← c + d
  - x₀ ← e + f
- Node G
  - r₂ ← φ(r₀,r₁)
  - y₀ ← a + b
  - z₀ ← c + d
Computing Dominators

Critical first step in SSA construction and in DVNT

• A node \( n \) dominates \( m \) iff \( n \) is on every path from \( n_0 \) to \( m \)
  ◦ Every node dominates itself
  ◦ \textit{n’s immediate dominator} is its closest dominator, \( \text{IDOM}(n) \)

\[
\text{DOM}(n_0) = \{ n_0 \}
\]
\[
\text{DOM}(n) = \{ n \} \cup \left( \cap_{p \in \text{preds}(n)} \text{DOM}(p) \right)
\]

Computing DOM

• These simultaneous set equations the data-flow problem
  ◦ The simplest equations we have seen
  ◦ Transfer function is the identity function

• Equations have a unique fixed point solution
• An iterative fixed-point algorithm will solve them quickly

\( \text{IDOM}(n) \neq n, \text{ unless } n \text{ is } n_0, \text{ by convention.} \)
Round-robin Iterative Algorithm for DOM

\[
\begin{align*}
\text{DOM}(n_0) & \leftarrow n_0 \\
\text{for } x & \leftarrow n_1 \text{ to } n_n \\
& \quad \text{DOM}(x) \leftarrow \{ \text{all nodes in graph} \} \\
\text{change} & \leftarrow \text{true} \\
\text{while (change)} \\
& \quad \text{change} \leftarrow \text{false} \\
& \quad \text{for } x \leftarrow n_0 \text{ to } n_n \\
& \quad & \quad \text{TEMP} \leftarrow \{ x \} \cup \left( \cap_{y \in \text{pred}(x)} \text{DOM}(y) \right) \\
& \quad & \quad \text{if } \text{DOM}(x) \neq \text{TEMP} \text{ then} \\
& \quad & \quad \quad \text{change} \leftarrow \text{true} \\
& \quad & \quad \quad \text{DOM}(x) \leftarrow \text{TEMP}
\end{align*}
\]

Termination

- Makes sweeps over the nodes
- Halts when some sweep produces no change
DOM Example

Flow Graph

Progress of iterative solution for DOM

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
</tbody>
</table>
Example

Progress of iterative solution for DOM

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
</tbody>
</table>

Results of iterative solution for DOM

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>IDOM</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

There are asymptotically faster algorithms.

With the right data structures, the iterative algorithm can be made extremely fast (competitive out to 10,000 or 20,000 nodes)

See Cooper, Harvey, & Kennedy [100], or § 9.5.2 in EaC2e.
Proliferation of GDFAPs

In the late 1960’s and the 1970’s many data-flow problems were proposed

• GDFAP became the standard way to prove safety of a transformation
  ♦ New transformation implied new GDFAP
  ♦ Optimizing compilers spent a large fraction of compile time solving GDFAPs
  ♦ Computers were relatively slow (1 – 10 MIPS) and small (16 to 32 MB)
  ♦ Development of “frameworks” for G DFA

• As transformations proliferated, need for a new paradigm emerged
  ♦ One GDFAP that could be used for multiple transformations
  ♦ Simplify the implementation
  ♦ Reduce the time spent in analysis
  ♦ The result was the development of information chains

In truth, the story is not that simple. Information chains did not arise overnight in response to an excessive number of GDFAPs; however, by the late 1980’s they were being used to replace individual GDFAPs.
**Information Chains**

**A tuple that connects 2 data-flow events is a chain**

- Chains express data-flow relationships directly.
- Chains provide a graphical representation.
- Chains jump across unrelated code, simplifying search.

We can build chains efficiently.

Four interesting types of chain

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Dependence Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>USE</td>
<td><em>true, flow</em></td>
</tr>
<tr>
<td>USE</td>
<td>DEF</td>
<td><em>anti</em></td>
</tr>
<tr>
<td>DEF</td>
<td>DEF</td>
<td><em>output</em></td>
</tr>
<tr>
<td>USE</td>
<td>USE</td>
<td><em>input</em></td>
</tr>
</tbody>
</table>

*DEF-USE* chains are the most common.
Information Chains

Example

```
a ← 5
b ← 3
c ← b + 2
d ← a - 2
```

```
e ← a + b
e ← e + c
```

```
f ← 2 + e
write f
```

```
e ← 13
```

**DEF-USE** Chains

**DEF-USE** Chains form a sparse evaluation graph that we can use in many transformations. For example, a **DEF** with no reachable use is *dead.*
Notation

Assume that, ∀ operation i and each variable v,

- \( \text{DEFS}(v,i) \) is the set of operations that may have defined \( v \) most recently before \( i \), along some path in the CFG
- \( \text{USES}(v,i) \) is the set of operations that may use the value of \( v \) computed at \( i \), along some path in the CFG

\[
x \in \text{DEFS}(A,y) \iff y \in \text{USES}(A,x)
\]

To construct DEF-USE chains, we solve reaching definitions \( \text{(YAGDFAP)} \)

- A definition \( d \) of some variable \( v \) reaches an operation \( i \) if and only if \( i \) reads \( v \) and there is a \( v\text{-clear} \) path from \( d \) to \( i \)
  - \( v\text{-clear} \Rightarrow \) no definition of \( v \) on the path
- Prior definition of \( v \) in same block \( \Rightarrow |\text{DEFS}(v,i)| = 1 \)
- No prior definition \( \Rightarrow |\text{DEFS}(v,i)| \geq 1 \)
Reaching Definitions

The equations

\[ \text{REACHES}(b) = \emptyset, \forall \ n \in N \]
\[ \text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} (\text{DEDEF}(p) \cup (\text{REACHES}(p) \cap \text{DEFKILL}(p))) \]

- \( \text{REACHES}(b) \) is the set of definitions that reach block \( b \)
- \( \text{DEDEF}(b) \) is the set of definitions in \( n \) that reach the end of \( b \)
- \( \text{DEFKILL}(b) \) is the set of defs obscured by a new def in \( b \)

Computing \( \text{REACHES}(b) \)

- Use any data-flow method \( \text{(rapid framework)} \)
- Zadeck shows a simple linear-time algorithm

Building DEFS Sets

The Plan

1. Find basic blocks & build the CFG
2. ∀ block \( b \), compute \( \text{REACHES}(b) \)

3. ∀ block \( b \), ∀ operation \( i \), ∀ referenced name \( v \),
   Set \( \text{DEFS}(v,i) \) according to the earlier rule
   
   If there is a prior definition, \( d \), of \( v \) in \( b \)
   
   \[ \text{DEFS}(v,i) \leftarrow d \]

   Otherwise
   
   \[ \text{DEFS}(v,i) \leftarrow \{ d \mid d \text{ defines } v \text{ & } d \in \text{REACHES}(b) \} \]

To build USES

• Invert \( \text{DEFS} \), or

• Solve *reachable uses*, and use the analogous construction
Building DEF-USE Chains

Miscellany

• Domain of \textbf{REACHES} is the set of definitions \((\propto |\text{operations}|)\)
• Domain of \textbf{DEFS} & \textbf{USES} is also definitions
• Need a compact representation of \textbf{DEFS} & \textbf{USES}

Detecting Anomalies

• \textbf{DEFS}(v,i) = \emptyset \text{ implies use of a never initialized variable}
• Add a definition for each \(v\) to \(n_0\) to detect larger set of anomalies
  ♦ If initial \(\text{def} \in \textbf{DEFS}(v,i)\) then \(\exists\) a path to \(i\) with no initialization

\textbf{NEXT LECTURE}: using information chains & moving into SSA
SVN did not help with blocks F or G

- Multiple predecessors

- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging state is expensive
  - Fall back on what’s known

- Can use table from $\text{idom}(x)$ to start $x$
  - Use C for F and A for G
  - Imposes a DOM-based application order

Leads to Dominator VN Technique (DVNT)
Dominator Value Numbering

The DVNT Algorithm

• Use superlocal algorithm on extended basic blocks
  ♦ Retain use of scoped hash tables
  ♦ Need to use the SSA name space to avoid bookkeeping headaches

• Start each node with table from its IDOM
  ♦ DVNT generalizes the superlocal algorithm

• No values flow along back edges \((i.e., \text{around loops})\)

• Constant folding, algebraic identities as before

Larger scope leads to \((potentially)\) better results

♦ LVN + SVN + good start for EBBs missed by SVN
**Dominator Value Numbering**

**DVNT advantages**
- Find more redundancy
- Little additional cost
- Retains *online* character

**DVNT shortcomings**
- Misses some opportunities
- No loop-carried redundancies or constants

See [53] or § 10.5.2 in EaC2e