Using Program Analysis for Optimization

Analysis and Optimizations

• Program Analysis
  – Discovers properties of a program

• Optimizations
  – Use analysis results to transform program
  – Goal: improve some aspect of program
    • number of executed instructions, number of cycles
    • cache hit rate
    • memory space (code or data)
    • power consumption
  – Has to be safe: Keep the semantics of the program

Control Flow Graph

<table>
<thead>
<tr>
<th>entry</th>
<th>s = 0; a = 4; i = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if (k == 0) b = 1;</td>
</tr>
<tr>
<td></td>
<td>else b = 2;</td>
</tr>
<tr>
<td></td>
<td>while (i &lt; n) {</td>
</tr>
<tr>
<td></td>
<td>s = s + a*b;</td>
</tr>
<tr>
<td></td>
<td>i = i + 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>return s;</td>
</tr>
</tbody>
</table>

Control Flow Graph

• Nodes represent computation
  – Each node is a Basic Block
  – A Basic Block is a sequence of instructions with
    • No branches out of middle of basic block
    • No branches into middle of basic block
    • Basic blocks should be maximal
  – Execution of basic block starts with first instruction
  – Includes all instructions in basic block
• Edges represent control flow

Two Kinds of Variables

• Temporaries introduced by the compiler
  – Transfer values only within basic block
  – Introduced as part of instruction flattening
  – Introduced by optimizations/transformations

• Program variables
  – Declared in original program
  – May transfer values between basic blocks

Basic Block Optimizations

• Common Sub-Expression Elimination
  – a = (x+y)z; b = x+y;
  – t = x+y; a = t+z; b = t;

• Copy Propagation
  – a = x+y; b = a; c = b+z;
  – a = x+y; b = a; c = a+z;

• Constant Propagation
  – x = 5; b = x+y;
  – b = 5+y;

• Dead Code Elimination
  – a = x+y; b = a; c = a+z;
  – a = x+y; c = a+z

• Algebraic Simplification
  – a = x * 1;
  – a = x;

• Strength Reduction
  – t = i * 4;
  – t = i << 2;
Value Numbering

• Normalize basic block so that all statements are of the form
  – var = var op var (where op is a binary operator)
  – var = op var (where op is a unary operator)
  – var = var

• Simulate execution of basic block
  – Assign a virtual value to each variable
  – Assign a virtual value to each expression
  – Assign a temporary variable to hold value of each computed expression

Value Numbering for CSE

• As we simulate execution of program
  • Generate a new version of program
    – Each new value assigned to temporary
      a = x+y; becomes a = x+y; t = a;
      – Temporary preserves value for use later in program even if original variable is rewritten
        a = x+y; a = a+z; b = x+y
        becomes
        a = x+y; t = a; a = a+z; b = t;

CSE Example

• Original
  a = x+y
  b = a+z
  b = b+y
  c = a+z

• After CSE
  a = x+y
  b = a+z
  t = b
  b = b+y
  c = t

• Issues
  – Temporaries store values for use later
  – CSE with different names
    a = x; b = x+y; c = a+y;
  – Excessive generation and use of temporaries

Problems

• Algorithm has a temporary for each new value
  – a = x+y; t1 = a

• Introduces
  – lots of temporaries
  – lots of copy statements to temporaries

• In many cases, temporaries and copy statements are unnecessary
  So we eliminate them with copy propagation and dead code elimination

Copy Propagation

• Once again, simulate execution of program
  • If possible, use the original variable instead of a temporary
    – a = x+y; b = x+y;
    – After CSE becomes a = x+y; t = a; b = t;
    – After CP becomes a = x+y; t = a; b = a;
  • Key idea: determine when original variables are NOT overwritten between computation of stored value and use of stored value
Copy Propagation Maps

- Maintain two maps
  - tmp to var: tells which variable to use instead of a given temporary variable
  - var to set (inverse of tmp to var): tells which temps are mapped to a given variable by tmp to var

Copy Propagation Example

<table>
<thead>
<tr>
<th>Original</th>
<th>After CSE</th>
<th>After CSE and Copy Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x+y</td>
<td>a = x+y</td>
<td>a = x+y</td>
</tr>
<tr>
<td>b = a+z</td>
<td>t1 = a</td>
<td>t1 = a</td>
</tr>
<tr>
<td>c = x+y</td>
<td>b = a+z</td>
<td>b = a+z</td>
</tr>
<tr>
<td>a = b</td>
<td>t2 = b</td>
<td>t2 = b</td>
</tr>
</tbody>
</table>

Copy Propagation Example

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>Basic Block After CSE</th>
<th>Basic Block After CSE and Copy Prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = x+y</td>
<td>a = x+y</td>
<td>a = x+y</td>
</tr>
<tr>
<td>t1 = a</td>
<td>t1 = a</td>
<td>t1 = a</td>
</tr>
<tr>
<td>b = a+z</td>
<td>b = a+z</td>
<td>b = a+z</td>
</tr>
<tr>
<td>t2 = b</td>
<td>t2 = b</td>
<td>t2 = b</td>
</tr>
</tbody>
</table>

tmp to var: t1 → a  var to set: a → {t1}
Copy Propagation Example

Basic Block
After CSE

Basic Block
After CSE and Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = t_1 \]
\[ a = b \]
\[ \text{tmp to var} \rightarrow \text{var to set} \]
\[ t_1 \rightarrow a \]
\[ t_2 \rightarrow b \]

Basic Block After CSE

Basic Block After CSE and Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = t_1 \]
\[ a = b \]
\[ \text{tmp to var} \rightarrow \text{var to set} \]
\[ t_1 \rightarrow t_1 \]
\[ t_2 \rightarrow t_2 \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop + DCE

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]

Basic Block After CSE + Copy Prop

Basic Block After CSE + Copy Prop

\[ a = x + y \]
\[ t_1 = a \]
\[ b = a + z \]
\[ t_2 = b \]
\[ c = a \]
\[ a = b \]
Basic Block After
CSE + Copy Propagation

\[
a = x + y \\
t_1 = a \\
b = a + z \\
t_2 = b \\
c = a \\
a = b
\]

Needed Set
\{b, c\}

Basic Block After
CSE + Copy Propagation

\[
a = x + y \\
t_1 = a \\
b = a + z \\
t_2 = b \\
c = a \\
a = b
\]

Needed Set
\{a, b\}

Basic Block after
CSE + Copy Propagation + Dead Code Elimination

\[
a = x + y \\
t_1 = a \\
b = a + z \\
c = a \\
a = b
\]

Needed Set
\{a, b\}

Basic Block after
CSE + Copy Propagation + Dead Code Elimination

\[
a = x + y \\
t_1 = a \\
b = a + z \\
c = a \\
a = b
\]

Needed Set
\{a, b, z\}

Basic Block after
CSE + Copy Propagation + Dead Code Elimination

\[
a = x + y \\
t_1 = a \\
b = a + z \\
c = a \\
a = b
\]

Needed Set
\{a, z\}
Interesting Properties

- Analysis and Optimization algorithms simulate execution of program
  - CSE and Copy Propagation go forward
  - Dead Code Elimination goes backwards
- Optimizations are stacked
  - Group of basic transformations
  - Work together to get good result
  - Often, one transformation creates inefficient code that is cleaned up by subsequent transformations

Other Basic Block Transformations

- Constant Propagation
- Strength Reduction
  - $a \ll 2 = a \times 4$;
  - $a + a + a = 3 \times a$;
- Algebraic Simplification
  - $a = a \times 1$;
  - $b = b + 0$;
- Need a unified transformation framework

Dataflow Analysis

- Used to determine properties of programs that involve multiple basic blocks
- Typically used to enable transformations
  - common sub-expression elimination
  - constant and copy propagation
  - dead code elimination
- Analysis and transformation often come in pairs

Reaching Definitions

- Concept of definition and use
  - $z = x + y$
  - is a definition of $z$
  - is a use of $x$ and $y$
- A definition reaches a use if
  - value written by definition
  - may be read by use
Reaching Definitions

\[
\begin{align*}
\text{s} &= 0; \\
a &= 4; \\
i &= 0; \\
k &= 0 \\
b &= 1; \\
b &= 2; \\
i < n \
\end{align*}
\]

\[i = i + 1; \rightarrow s = s + a \times b; \rightarrow \text{return } s\]

Reaching Definitions and Constant Propagation

• Is a use of a variable a constant?
  – Check all reaching definitions
  – If all assign variable to same constant
  – Then use is in fact a constant
• Can replace variable with constant

Is a constant in \(s = s + a \times b\)?

\[
\begin{align*}
s &= 0; \\
a &= 4; \\
i &= 0; \\
k &= 0 \\
b &= 1; \\
b &= 2; \\
i < n \
\end{align*}
\]

\[i = i + 1; \rightarrow s = s + a \times b; \rightarrow \text{return } s\]

Yes!
On all reaching definitions
\(a = 4\)

Constant Propagation Transform

\[
\begin{align*}
s &= 0; \\
a &= 4; \\
i &= 0; \\
k &= 0 \\
b &= 1; \\
b &= 2; \\
i < n \
\end{align*}
\]

\[i = i + 1; \rightarrow s = s + a \times b; \rightarrow \text{return } s\]

Yes!
On all reaching definitions
\(a = 4\)

Is \(b\) constant in \(s = s + a \times b\)?

\[
\begin{align*}
s &= 0; \\
a &= 4; \\
i &= 0; \\
k &= 0 \\
b &= 1; \\
b &= 2; \\
i < n \
\end{align*}
\]

\[i = i + 1; \rightarrow s = s + a \times b; \rightarrow \text{return } s\]

No!
One reaching definition with \(b = 1\)
One reaching definition with \(b = 2\)

Computing Reaching Definitions

• Compute with sets of definitions
  – represent sets using bit vectors
  – each definition has a position in bit vector
• At each basic block, compute
  – definitions that reach start of block
  – definitions that reach end of block
• Do computation by simulating execution of program until the fixed point is reached
Formalizing Analysis

- Each basic block has
  - **IN** - set of definitions that reach beginning of block
  - **OUT** - set of definitions that reach end of block
  - **GEN** - set of definitions generated in block
  - **KILL** - set of definitions killed in the block
- \( \text{GEN}[s = s + a*b; i = i + 1;] = 0000011 \)
- \( \text{KILL}[s = s + a*b; i = i + 1;] = 1010000 \)
- Compiler scans each basic block to derive **GEN** and **KILL** sets

Dataflow Equations

- \( \text{IN}[b] = \text{OUT}[b1] \cup \ldots \cup \text{OUT}[bn] \)
  - where \( b1, \ldots, bn \) are predecessors of \( b \) in CFG
- \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- \( \text{IN}[\text{entry}] = 0000000 \)
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- Initialize with solution of \( \text{OUT}[b] = 0000000 \)
- Repeatedly apply equations
  - \( \text{IN}[b] = \text{OUT}[b1] \cup \ldots \cup \text{OUT}[bn] \)
  - \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- Until reaching fixed point
  - i.e., until equation application has no further effect
- Use a worklist to track which equation applications may have a further effect

Reaching Definitions Algorithm

for all nodes \( n \) in \( N \) \( \text{OUT}[n] = \emptyset \); // \( \text{OUT}[n] = \text{GEN}[n] \);
Worklist = \( N \); // \( N \) = all nodes in graph
while (Worklist != \( \emptyset \))
  choose a node \( n \) in Worklist;
  Worklist = Worklist - \{ \( n \) \};
  \( \text{IN}[n] = \emptyset \);
  for all nodes \( p \) in predecessors(\( n \)) \( \text{IN}[n] = \text{IN}[n] \cup \text{OUT}[p] \);
  \( \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \);
  if (\( \text{OUT}[n] \) changed)
    for all nodes \( s \) in successors(\( n \)) Worklist = Worklist \cup \{ \( s \) \}.
Questions

- Does the algorithm halt?
  - yes, because transfer function is monotonic
  - if increase IN, increase OUT
  - in limit, all bits are 1
- If bit is 1, is there always an execution in which corresponding definition reaches basic block?
- If bit is 0, does the corresponding definition ever reach basic block?
- Concept of conservative analysis

Available Expressions

- An expression x+y is available at a point p if
  - every path from the initial node to p evaluates x+y before reaching p
  - and there are no assignments to x or y after the evaluation but before p
- Available Expression information can be used to do global (across basic blocks) CSE
- If an expression is available at use, there is no need to re-evaluate it

Computing Available Expressions

- Represent sets of expressions using bit vectors
- Each expression corresponds to a bit
- Run dataflow algorithm similar to reaching definitions
- Big difference:
  - A definition reaches a basic block if it comes from ANY predecessor in CFG
  - An expression is available at a basic block only if it is available from ALL predecessors in CFG

Available Expressions Example

Expressions
1: x+y
2: i < n
3: i+c
4: x == 0

Global CSE Transform

Expressions
1: x+y
2: i < n
3: i+c
4: x == 0

must use same temp for CSE in all blocks

Formalizing Analysis

- Each basic block has
  - IN - set of expressions available at start of block
  - OUT - set of expressions available at end of block
  - GEN - set of expressions computed in block
  - KILL - set of expressions killed in the block
- GEN[x = z; b = x+y] = 1000
- KILL[x = z; b = x+y] = 1001
- Compiler scans each basic block to derive GEN and KILL sets
**Dataflow Equations**

- \( \text{IN}[b] = \text{OUT}[b1] \cap ... \cap \text{OUT}[bn] \)
  - where \( b1, ..., bn \) are predecessors of \( b \) in CFG
- \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- \( \text{IN}[\text{entry}] = 0000 \)
- Result: system of equations

**Solving Equations**

- Use fixed point algorithm
- \( \text{IN}[\text{entry}] = 0000 \)
- Initialize \( \text{OUT}[b] = 1111 \)
- Repeatedly apply equations
  - \( \text{IN}[b] = \text{OUT}[b1] \cap ... \cap \text{OUT}[bn] \)
  - \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- Use a worklist algorithm to track which equation applications may have further effect

**Available Expressions Algorithm**

\[
\begin{align*}
\text{for all nodes } n \text{ in } N & : \text{OUT}[n] = E; \quad \text{// OUT}[n] = E - \text{KILL}[n]; \\
\text{IN[Entry]} = \emptyset; \text{OUT[Entry]} = \text{GEN[Entry]}; \\
\text{Worklist} = N - \{ \text{Entry} \}; \quad \text{// N = all nodes in graph} \\
\text{while (Worklist} \neq \emptyset) & : \text{// choose a node } n \text{ in Worklist;} \\
\text{Worklist} = \text{Worklist} - \{ n \}; \quad \text{// worklist unchanged} \\
\text{IN}[n] = E; \quad \text{// E is set of all expressions} \\
\text{for all nodes } p \text{ in predecessors}(n) & : \text{IN}[n] = \text{IN}[n] \cap \text{OUT}[p]; \\
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]; \quad \text{if (OUT}[n] \text{ changed)} \\
\text{for all nodes } s \text{ in successors}(n) & : \text{Worklist} = \text{Worklist} \cup \{ s \}; \\
\end{align*}
\]

**Questions**

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?
- In what sense is the algorithm conservative?

**Duality In Two Algorithms**

- Reaching definitions
  - Confluence operation is set union
  - \( \text{OUT}[b] \) initialized to empty set
- Available expressions
  - Confluence operation is set intersection
  - \( \text{OUT}[b] \) initialized to set of available expressions
- General framework for dataflow algorithms
- Build parameterized dataflow analyzer once, use for all dataflow analysis problems

**Liveness Analysis**

- A variable \( v \) is live at point \( p \) if
  - \( v \) is used along some path starting at \( p \), and
  - no definition of \( v \) along the path before the use.
- When is a variable \( v \) dead at point \( p \)?
  - No use of \( v \) on any path from \( p \) to exit node, or
  - If all paths from \( p \), redefine \( v \) before using \( v \).
What Use is Liveness Information?

- **Register allocation**
  - If a variable is dead, we can reassign its register
- **Dead code elimination**
  - Eliminate assignments to variables not read later
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG
  - Can eliminate other dead assignments
  - Handle by making all externally visible variables live on exit from CFG

Conceptual Idea of Analysis

- **Simulate execution**
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks

Liveness Example

- Assume a, b, c visible outside function and thus are live on exit
- Assume x, y, z, t are not visible on exit
- Represent liveness using a bit vector
  - order is abcxzyt

Using Liveness Information for Dead Code Elimination

- Assume a, b, c visible outside function and thus are live on exit
- Assume x, y, z, t are not visible on exit
- Represent liveness using a bit vector
  - order is abcxzyt

Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block
  - OUT - set of variables live at end of block
  - USE - set of variables with upwards exposed uses in block
  - DEF - set of variables defined in block
- **USE**\[x = z; x = x + 1;\] = \{ z \} (x not in USE)
- **DEF**\[x = z; x = x + 1; y = 1;\] = \{ x, y \}
- Compiler scans each basic block to derive USE and DEF sets

Algorithm

\[
\text{OUT}[	ext{Exit}] = \emptyset; \\
\text{IN}[	ext{Exit}] = \text{USE}[n]; \\
\text{for all nodes } n \text{ in } N - \{ \text{ Exit } \} \text{ IN}[n] = \emptyset; \\
\text{Worklist} = N - \{ \text{ Exit } \}; \\
\text{while} (\text{Worklist} \neq \emptyset) \\
\text{choose a node } n \text{ in Worklist}; \\
\text{Worklist} = \text{Worklist} - \{ n \}; \\
\text{OUT}[n] = \emptyset; \\
\text{for all nodes } s \text{ in successors}(n) \text{ OUT}[n] = \text{OUT}[n] \cup \text{IN}[s]; \\
\text{IN}[n] = \text{USE}[n] \cup (\text{OUT}[n] - \text{DEF}[n]); \\
\text{if} (\text{IN}[n] \text{ changed}) \\
\text{for all nodes } p \text{ in predecessors}(n) \text{ Worklist} = \text{Worklist} \cup \{ p \};
\]
Similar to Other Dataflow Algorithms

- Backwards analysis, not forwards
- Still have transfer functions
- Still have confluence operators
- Can generalize framework to work for both forwards and backwards analyses

Analysis Information Inside Basic Blocks

- One detail:
  - Given dataflow information at IN and OUT of node
  - Also need to compute information at each statement of basic block
  - Simple propagation algorithm usually works fine
  - Can be viewed as restricted case of dataflow analysis

Summary

- Basic blocks and basic block optimizations
  - Copy and constant propagation
  - Common sub-expression elimination
  - Dead code elimination
- Dataflow Analysis
  - Control flow graph
  - IN[b], OUT[b], transfer functions, join points
- Paired of analyses and transformations
  - Reaching definitions/constant propagation
  - Available expressions/common sub-expression elimination
  - Liveness analysis/Dead code elimination