Introduction to Dataflow Analysis
Value Numbering Summary

- Forward symbolic execution of basic block
- Maps
  - Var2Val – symbolic value for each variable
  - Exp2Val – value of each evaluated expression
  - Exp2Tmp – tmp that holds value of each evaluated expression
- Algorithm
  - For each statement
    - If variables in RHS not in the Var2Val add it with a new value
    - If RHS expression in Exp2Tmp use that Temp
    - If not add RHS expression to Exp2Val with new value
    - Copy the value into a new tmp and add to Exp2Tmp
Copy Propagation Summary

- Forward Propagation within basic block
- Maps
  - \( \text{tmp2var} \): tells which variable to use instead of a given temporary variable
  - \( \text{var2set} \): inverse of \text{tmp} to \text{var}. tells which temps are mapped to a given variable by \text{tmp} to \text{var}
- Algorithm
  - For each statement
    - If any \text{tmp} variable in the RHS is in \text{tmp2var} replace it with \text{var}
    - If LHS \text{var} in \text{var2set} remove the variables in the set in \text{tmp2var}
Dead Code Elimination Summary

- Backward Propagation within basic block
- Map
  - A set of variables that are needed later in computation
- Algorithm
  - Every statement encountered
    - If LHS is not in the set, remove the statement
    - Else put all the variables in the RHS into the set
Summary So far... what's next

• Till now: How to analyze and transform within a basic block

• Next: How to do it for the entire procedure
Outline

- Reaching Definitions
- Available Expressions
- Liveness
Reaching Definitions

• Concept of definition and use
  - \( a = x+y \)
  - is a definition of \( a \)
  - is a use of \( x \) and \( y \)

• A definition reaches a use if
  - value written by definition
  - may be read by use
Reaching Definitions

\[
\begin{align*}
  s &= 0; \\
  a &= 4; \\
  i &= 0; \\
  k &= 0
\end{align*}
\]

\[
\begin{align*}
  b &= 1; \\
  b &= 2;
\end{align*}
\]

\[
\begin{align*}
  i &= n \\
  s &= s + a \times b; \\
  i &= i + 1;
\end{align*}
\]

\[
\text{return } s
\]
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 \( s = s + a \times b \) ?

Yes!

On all reaching definitions

\( a = 4 \)
Constant Propagation Transform

s = 0;
a = 4;
i = 0;
k == 0

b = 1;
b = 2;
i < n

s = s + 4*b;
i = i + 1;

return s

Yes!
On all reaching definitions
a = 4
Is \( b \) Constant in \( s = s + a \times b \)?

No!

One reaching definition with \( b = 1 \)

One reaching definition with \( b = 2 \)

\[
\begin{align*}
&\text{s = 0; } \\
&a = 4; \\
&i = 0; \\
&k == 0 \\
&\text{b = 1;} \\
&\text{b = 2;} \\
&i < n \\
&s = s + a \times b; \\
&i = i + 1; \\
\text{return s}
\end{align*}
\]
Splitting Preserves Information Lost At Merges

\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k == 0 \]

\[ b = 1; \]
\[ b = 2; \]

\[ i < n \]

\[ s = s + a \times b; \]
\[ i = i + 1; \]

\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k == 0 \]

\[ b = 1; \]
\[ b = 2; \]

\[ i < n \]

\[ s = s + a \times b; \]
\[ i = i + 1; \]

\[ return \ s \]
Splitting
Preserves Information Lost At Merges

\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k == 0 \]

\[ b = 1; \]
\[ b = 2; \]

\[ i < n \]

\[ s = s + a \times b; \]
\[ i = i + 1; \]

\[ \text{return } s \]

\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k == 0 \]

\[ b = 1; \]
\[ b = 2; \]

\[ i < n \]

\[ s = s + a \times 1; \]
\[ i = i + 1; \]

\[ \text{return } s \]

\[ s = s + a \times 2; \]
\[ i = i + 1; \]

\[ \text{return } s \]
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 reach fixed point
1: \( s = 0; \)
2: \( a = 4; \)
3: \( i = 0; \)
4: \( b = 1; \)
5: \( b = 2; \)
6: \( s = s + a \times b; \)
7: \( i = i + 1; \)

\( k == 0 \)

\( i < n \)

return \( s \)
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 block
- GEN\[s = s + a*b; i = i + 1;]\] = 0000011
- KILL\[s = s + a*b; i = i + 1;]\] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets
Dataflow Equations

- \( \text{IN}[b] = \text{OUT}[b_1] \cup ... \cup \text{OUT}[b_n] \)
  - where \( b_1, ..., b_n \) 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 OUT[b] = 0000000
- Repeatedly apply equations
  - IN[b] = OUT[b1] U ... U OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Until reach fixed point
- 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
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { 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 0, does the corresponding definition ever reach basic block?

• If bit is 1, is does the corresponding definition always reach the basic block?
1: s = 0;
2: a = 4;
3: i = 0;
4: b = 1;
5: b = 2;
6: s = s + a*b;
7: i = i + 1;

i < n

return s
Outline

• Reaching Definitions
• Available Expressions
• Liveness
Available Expressions

- An expression $x+y$ is available at a point $p$ if
  - every path from the initial node to $p$ must evaluate $x+y$ 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 expression is available at use, no need to reevaluate it
Example: Available Expression

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

\[ g = a + c \]

\[ b = a + d \]
\[ h = c + f \]

\[ j = a + b + c + d \]
Is the Expression Available?

YES!

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

\[ g = a + c \]
\[ b = a + d \]
\[ h = c + f \]

\[ j = a + b + c + d \]
Is the Expression Available?

YES!

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]
\[ g = a + c \]
\[ j = a + b + c + d \]
\[ b = a + d \]
\[ h = c + f \]
Is the Expression Available?

a = b + c

\[ d = e + f \]

f = a + c

\[ g = a + c \]

b = a + d

\[ h = c + f \]

\[ j = a + b + c + d \]

NO!
Is the Expression Available?

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

\[ g = a + c \]

\[ b = a + d \]
\[ h = c + f \]

\[ j = a + b + c + d \]

NO!
Is the Expression Available?

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

\[ g = a + c \]
\[ b = a + d \]
\[ h = c + f \]

\[ j = a + b + c + d \]

\textit{NO!}
Is the Expression Available?

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

YES!

\[ g = a + c \]
\[ b = a + d \]
\[ h = c + f \]

\[ j = a + b + c + d \]
Is the Expression Available?

YES!

\[
\begin{align*}
a &= b + c \\
d &= e + f \\
f &= a + c \\
g &= a + c \\
b &= a + d \\
h &= c + f \\
j &= a + b + c + d
\end{align*}
\]
Use of Available Expressions

\[
\begin{align*}
    a &= b + c \\
    d &= e + f \\
    f &= a + c \\
    g &= a + c \\
    b &= a + d \\
    h &= c + f \\
    j &= a + b + c + d
\end{align*}
\]
Use of Available Expressions

\[
\begin{align*}
a &= b + c \\
d &= e + f \\
f &= a + c \\
g &= a + c \\
b &= a + d \\
h &= c + f \\
j &= a + b + c + d
\end{align*}
\]
Use of Available Expressions

\[
\begin{align*}
    a &= b + c \\
    d &= e + f \\
    f &= a + c \\
    g &= a + c \\
    b &= a + d \\
    h &= c + f \\
    j &= a + b + c + d
\end{align*}
\]
Use of Available Expressions

\[
\begin{align*}
  a &= b + c \\
  d &= e + f \\
  f &= a + c \\
  g &= f \\
  b &= a + d \\
  h &= c + f \\
  j &= a + b + c + d
\end{align*}
\]
Use of Available Expressions

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]
\[ g = f \]
\[ b = a + d \]
\[ h = c + f \]
\[ j = a + b + c + d \]
Use of Available Expressions

\[
\begin{align*}
\text{a} & = \text{b} + \text{c} \\
\text{d} & = \text{e} + \text{f} \\
\text{f} & = \text{a} + \text{c}
\end{align*}
\]

\[
\begin{align*}
\text{g} & = \text{f} \\
\text{b} & = \text{a} + \text{d} \\
\text{h} & = \text{c} + \text{f}
\end{align*}
\]

\[
\text{j} = \text{a} + \text{c} + \text{b} + \text{d}
\]
Use of Available Expressions

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]

\[ j = f + b + d \]

\[ g = f \]

\[ b = a + d \]
\[ h = c + f \]
Use of Available Expressions

\[
\begin{align*}
a &= b + c \\
d &= e + f \\
f &= a + c
\end{align*}
\]

\[
\begin{align*}
g &= f \\
b &= a + d \\
h &= c + f
\end{align*}
\]

\[
\begin{align*}
j &= f + b + d
\end{align*}
\]
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
  - definition reaches a basic block if it comes from *any* predecessor in CFG
  - expression is available at a basic block only if it is available from *all* predecessors in CFG
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
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 in block
- \( \text{GEN}[x = z; b = x+y] = 1000 \)
- \( \text{KILL}[x = z; b = x+y] = 1001 \)
- 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
• \( \text{IN[entry]} = 0000 \)
• 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 reach fixed point
Available Expressions
Algorithm

for all nodes n in N
  OUT[n] = E; // E is set of all expressions
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
  choose a node n in Changed;
  Changed = Changed - { n };

  IN[n] = E; // E is set of all expressions
  for all nodes p in predecessors(n)
    IN[n] = IN[n] \cap OUT[p];

  OUT[n] = GEN[n] \cup (IN[n] - KILL[n]);

  if (OUT[n] changed)
    for all nodes s in successors(n)
      Changed = Changed U { s };
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?
Duality In Two Algorithms

- Reaching definitions
  - Confluence operation is set union
  - OUT[b] initialized to empty set

- Available expressions
  - Confluence operation is set intersection
  - OUT[b] initialized to set of available expressions

- General framework for dataflow algorithms.
- Build parameterized dataflow analyzer once, use for all dataflow problems
Outline

• Reaching Definitions
• Available Expressions
• Liveness
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, 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 method
- So are live on exit
- Assume x, y, z, t not visible
- Represent Liveness Using Bit Vector
  - order is abcxyzt
Dead Code Elimination

- Assume $a,b,c$ visible outside method
- So are live on exit
- Assume $x,y,z,t$ not visible
- Represent Liveness Using Bit Vector
  - order is $abcxyzt$
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

for all nodes n in N - { Exit }
    \( \text{IN}[n] = \text{emptyset}; \)
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };;

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };;

    \( \text{OUT}[n] = \text{emptyset}; \)
    for all nodes s in successors(n)
        \( \text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[p]; \)

    IN[n] = use[n] U (out[n] - def[n]);

if (IN[n] changed)
    for all nodes p in predecessors(n)
        Changed = Changed U { 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
## Comparison

### Reaching Definitions

- for all nodes $n$ in $N$
  - $OUT[n] = \text{emptyset}$;
  - $IN[\text{Entry}] = \text{emptyset}$;
  - $OUT[\text{Entry}] = \text{GEN}[\text{Entry}]$;
  - $\text{Changed} = N - \{ \text{Entry} \}$;

- while ($\text{Changed} \neq \text{emptyset}$)
  - choose a node $n$ in $\text{Changed}$;
  - $\text{Changed} = \text{Changed} - \{ n \}$;

- $IN[n] = \text{emptyset}$;
- for all nodes $p$ in $\text{predecessors}(n)$
  - $IN[n] = IN[n] \cup OUT[p]$;
- $OUT[n] = \text{GEN}[n] \cup (IN[n] - \text{KILL}[n])$;
- if ($\text{OUT}[n]$ changed)
  - for all nodes $s$ in $\text{successors}(n)$
    - $\text{Changed} = \text{Changed} \cup \{ s \}$;

### Available Expressions

- for all nodes $n$ in $N$
  - $OUT[n] = E$;
  - $IN[\text{Entry}] = \text{emptyset}$;
  - $OUT[\text{Entry}] = \text{GEN}[\text{Entry}]$;
  - $\text{Changed} = N - \{ \text{Entry} \}$;

- while ($\text{Changed} \neq \text{emptyset}$)
  - choose a node $n$ in $\text{Changed}$;
  - $\text{Changed} = \text{Changed} - \{ n \}$;

- $IN[n] = E$;
- for all nodes $p$ in $\text{predecessors}(n)$
  - $IN[n] = IN[n] \cap OUT[p]$;
- $OUT[n] = \text{GEN}[n] \cup (IN[n] - \text{KILL}[n])$;
- if ($\text{OUT}[n]$ changed)
  - for all nodes $s$ in $\text{successors}(n)$
    - $\text{Changed} = \text{Changed} \cup \{ s \}$;

### Liveness

- for all nodes $n$ in $N - \{ \text{Exit} \}$
  - $IN[n] = \text{emptyset}$;
  - $OUT[\text{Exit}] = \text{emptyset}$;
  - $IN[\text{Exit}] = \text{use}[\text{Exit}]$;
  - $\text{Changed} = N - \{ \text{Exit} \}$;

- while ($\text{Changed} \neq \text{emptyset}$)
  - choose a node $n$ in $\text{Changed}$;
  - $\text{Changed} = \text{Changed} - \{ n \}$;

- $OUT[n] = \text{emptyset}$;
- for all nodes $s$ in $\text{successors}(n)$
  - $OUT[n] = OUT[n] \cup IN[p]$;
- $IN[n] = \text{use}[n] \cup (out[n] - \text{def}[n])$;
- if ($IN[n]$ changed)
  - for all nodes $p$ in $\text{predecessors}(n)$
    - $\text{Changed} = \text{Changed} \cup \{ p \}$;
### Comparison

#### Reaching Definitions

- for all nodes $n$ in $N$
  - $OUT[n] = \text{emptyset}$;
  - $IN[\text{Entry}] = \text{emptyset}$;
  - $OUT[\text{Entry}] = GEN[\text{Entry}]$;
  - $\text{Changed} = N - \{ \text{Entry} \}$;

- while ($\text{Changed} \neq \text{emptyset}$)
  - choose a node $n$ in $\text{Changed}$;
  - $\text{Changed} = \text{Changed} - \{ n \}$;

- $IN[n] = \text{emptyset}$;
- for all nodes $p$ in predecessors($n$)
  - $IN[n] = \text{IN}[n] \cup \text{OUT}[p]$;

- $OUT[n] = GEN[n] \cup (\text{IN}[n] - KILL[n])$;
- if ($\text{OUT}[n]$ changed)
  - for all nodes $s$ in successors($n$)
    - $\text{Changed} = \text{Changed} \cup \{ s \}$;

#### Available Expressions

- for all nodes $n$ in $N$
  - $OUT[n] = E$;
  - $IN[\text{Entry}] = \text{emptyset}$;
  - $OUT[\text{Entry}] = GEN[\text{Entry}]$;
  - $\text{Changed} = N - \{ \text{Entry} \}$;

- while ($\text{Changed} \neq \text{emptyset}$)
  - choose a node $n$ in $\text{Changed}$;
  - $\text{Changed} = \text{Changed} - \{ n \}$;

- $IN[n] = E$;
- for all nodes $p$ in predecessors($n$)
  - $IN[n] = \text{IN}[n] \cap \text{OUT}[p]$;

- $OUT[n] = GEN[n] \cup (\text{IN}[n] - KILL[n])$;
- if ($\text{OUT}[n]$ changed)
  - for all nodes $s$ in successors($n$)
    - $\text{Changed} = \text{Changed} \cup \{ s \}$;
## Comparison

### Reaching Definitions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes n in N</td>
<td></td>
</tr>
<tr>
<td>OUT[n] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>IN[Entry] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>OUT[Entry] = GEN[Entry];</td>
<td></td>
</tr>
<tr>
<td>Changed = N - { Entry };</td>
<td></td>
</tr>
<tr>
<td>while (Changed != emptyset)</td>
<td></td>
</tr>
<tr>
<td>choose a node n in Changed;</td>
<td></td>
</tr>
<tr>
<td>Changed = Changed - { n };</td>
<td></td>
</tr>
<tr>
<td>IN[n] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>for all nodes p in predecessors(n)</td>
<td></td>
</tr>
<tr>
<td>IN[n] = IN[n] U OUT[p];</td>
<td></td>
</tr>
<tr>
<td>OUT[n] = GEN[n] U (IN[n] - KILL[n]);</td>
<td></td>
</tr>
<tr>
<td>if (OUT[n] changed)</td>
<td></td>
</tr>
<tr>
<td>for all nodes s in successors(n)</td>
<td></td>
</tr>
<tr>
<td>Changed = Changed U { s };</td>
<td></td>
</tr>
</tbody>
</table>

### Liveness

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes n in N</td>
<td></td>
</tr>
<tr>
<td>IN[n] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>OUT[Exit] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>IN[Exit] = use[Exit];</td>
<td></td>
</tr>
<tr>
<td>Changed = N - { Exit };</td>
<td></td>
</tr>
<tr>
<td>while (Changed != emptyset)</td>
<td></td>
</tr>
<tr>
<td>choose a node n in Changed;</td>
<td></td>
</tr>
<tr>
<td>Changed = Changed - { n };</td>
<td></td>
</tr>
<tr>
<td>OUT[n] = emptyset;</td>
<td></td>
</tr>
<tr>
<td>for all nodes p in successors(n)</td>
<td></td>
</tr>
<tr>
<td>OUT[n] = OUT[n] U IN[p];</td>
<td></td>
</tr>
<tr>
<td>IN[n] = use[n] U (out[n] - def[n]);</td>
<td></td>
</tr>
<tr>
<td>if (IN[n] changed)</td>
<td></td>
</tr>
<tr>
<td>for all nodes p in predecessors(n)</td>
<td></td>
</tr>
<tr>
<td>Changed = Changed U { p };</td>
<td></td>
</tr>
</tbody>
</table>
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
Pessimistic vs. Optimistic Analyses

- Available expressions is optimistic (for common sub-expression elimination)
  - Assume expressions are available at start of analysis
  - Analysis eliminates all that are not available
  - Cannot stop analysis early and use current result
- Live variables is pessimistic (for dead code elimination)
  - Assume all variables are live at start of analysis
  - Analysis finds variables that are dead
  - Can stop analysis early and use current result
- Dataflow setup same for both analyses
- Optimism/pessimism depends on intended use
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 analyses and transformations
  – Reaching definitions/constant propagation
  – Available expressions/common sub-expression elimination
  – Liveness analysis/Dead code elimination

• Stacked analysis and transformations work together