Lecture 10: 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
  – tmp2var: tells which variable to use instead of a given temporary variable
  – var2set: inverse of tmp to var. tells which temps are mapped to a given variable by tmp to var

• Algorithm
  – For each statement
    • If any tmp variable in the RHS is in tmp2var replace it with var
    • If LHS var in var2set remove the variables in the set in 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
Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions
Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions

- Example

  - \( a + 0 \) \( \Rightarrow \) \( a \)
  - \( a * 1 \) \( \Rightarrow \) \( a \)
  - \( a / 1 \) \( \Rightarrow \) \( a \)
  - \( a * 0 \) \( \Rightarrow \) \( 0 \)
  - \( 0 - a \) \( \Rightarrow \) \( -a \)
  - \( a + (-b) \) \( \Rightarrow \) \( a - b \)
  - \( -(-a) \) \( \Rightarrow \) \( a \)
Algebraic Simplification

• Apply our knowledge from algebra, number theory etc. to simplify expressions

• Example
  - $a \land \text{true} \implies a$
  - $a \land \text{false} \implies \text{false}$
  - $a \lor \text{true} \implies \text{true}$
  - $a \lor \text{false} \implies a$
Algebraic Simplification

- Apply our knowledge from algebra, number theory etc. to simplify expressions

- Example
  - $a^2 \Rightarrow a \times a$
  - $a \times 2 \Rightarrow a + a$
  - $a \times 8 \Rightarrow a << 3$
Opportunities for Algebraic Simplification

• In the code
  – Programmers are lazy to simplify expressions
  – Programs are more readable with full expressions

• After compiler expansion
  – Example: Array read A[8][12] will get expanded to
  – *(Abase + 4*(12 + 8*256)) which can be simplified

• After other optimizations
Usefulness of Algebraic Simplification

- Reduces the number of instructions
- Uses less expensive instructions
- Enable other optimizations
Implementation

- Not a data-flow optimization!
- Find candidates that matches the simplification rules and simplify the expression trees
- Candidates may not be obvious
Implementation

- Not a data-flow optimization!
- Find candidates that matches the simplification rules and simplify the expression trees
- Candidates may not be obvious
  - Example
    \( a + b - a \)
Use knowledge about operators

- Commutative operators
  - $a \text{ op } b = b \text{ op } a$

- Associative operators
  - $(a \text{ op } b) \text{ op } c = b \text{ op } (a \text{ op } c)$
Canonical Format

• Put expression trees into a canonical format
  – Sum of multiplicands
  – Variables/terms in a canonical order
  – Example
    \((a+3)*(a+8)*4 \Rightarrow 4*a*a+44*a+96\)

  – Section 12.3.1 of whale book talks about this
Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  \[-(a / b)*0 + c\]
Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  - \((a / b) * 0 + c\)
  - we can simplify this to \(c\)
Effects on the Numerical Stability

- Some algebraic simplifications may produce incorrect results
- Example
  - \((a / b) \times 0 + c\)
  - we can simplify this to \(c\)
  - But what about when \(b = 0\)?
    should be a exception, but we’ll get a result!
Interesting Properties

• Analysis and Transformation Algorithms
  Symbolically Simulate Execution of Program
  – CSE and Copy Propagation go forward
  – Dead Code Elimination goes backwards

• Transformations stacked
  – Group of basic transformations work together
  – Often, one transformation creates inefficient code that is cleaned up by following transformations
  – Transformations can be useful even if original code may not benefit from transformation
Other Basic Block Transformations

- Constant Propagation
- Strength Reduction
  - \( a << 2 = a \times 4; a + a + a = 3 \times a; \)
- Do these in unified transformation framework, not in earlier or later phases
Summary So far...

- Basic block analyses and transformations
- Symbolically simulate execution of program
  - Forward (CSE, copy prop, constant prop)
  - Backward (Dead code elimination)
- Stacked groups of analyses and transformations that work together
  - CSE introduces excess temporaries and copy statements
  - Copy propagation often eliminates need to keep temporary variables around
  - Dead code elimination removes useless code
- Similar in spirit to many analyses and transformations that operate across basic blocks
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
  – \textbf{may} be read by use
Reaching Definitions

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

b = 1;
b = 2;
i < n

s = s + a*b;
i = i + 1;
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 \( s = s + a \times b \)?

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

Yes!

On all reaching definitions \( a = 4 \)
Constant Propagation Transform

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

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

No!

One reaching definition with $b = 1$

One reaching definition with $b = 2$
Splitting Preserves Information Lost At Merges

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

\[
b = 1; \\
b = 2; \\
i < n
\]

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

\[
return s
\]

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

\[
b = 1; \\
b = 2; \\
i < n
\]

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

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

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

i < n
6: s = s + a*b;
7: i = i + 1;

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

- \( \text{IN}[b] = \text{OUT}[b_1] \cup \ldots \cup \text{OUT}[b_n] \)
  - where \( b_1, \ldots, b_n \) 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 \( \text{OUT}[b] = 0000000 \)
- 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 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 \( N \)
   \( \text{OUT}[n] = \text{emptyset}; \) // \( \text{OUT}[n] = \text{GEN}[n] \);
\( \text{IN}[\text{Entry}] = \text{emptyset}; \)
\( \text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]; \)
\( \text{Changed} = N - \{ \text{Entry} \} ; \) // \( N \) = all nodes in graph

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

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

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

if (\( \text{OUT}[n] \) changed)
   for all nodes \( s \) in \( \text{successors}(n) \)
   \( \text{Changed} = \text{Changed} \cup \{ s \} ; \)
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 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: \( s = s + a \times b; \)
7: \( i = i + 1; \)

\( s = s + a \times b; \)

\( i < n \)

\( i = i + 1; \)

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!

\[
\begin{align*}
    a &= b + c \\
    d &= e + f \\
    f &= a + c \\
    g &= a + c \\
    b &= a + d \\
    h &= c + f \\
    j &= a + b + c + d
\end{align*}
\]
Is the Expression Available?

a = b + c

YES!

b = a + d

h = c + f

j = a + b + c + d

g = a + c

f = a + c

d = e + f
Is the Expression Available?

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

\textit{NO!}
Is the Expression Available?

a = b + c

NO!

d = e + f

f = a + c

g = a + c

b = a + d

h = c + f

j = a + b + c + d
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?

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?

a = b + c
b = a + d
c = f + e
f = a + c
g = a + c
h = c + f
j = a + b + c + d

YES!
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

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

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

\[ g = a + c \]

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

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

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

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

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

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

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

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

- \( \text{IN}[b] = \text{OUT}[b_1] \cap \ldots \cap \text{OUT}[b_n] \)
  - where \( b_1, \ldots, b_n \) are predecessors of \( b \) in CFG
- \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- \( \text{IN}[\text{entry}] = 0000 \)
- Result: system of equations
Solving Equations

- Use fixed point algorithm
- \( \text{IN[entry]} = 0000 \)
- Initialize \( \text{OUT[b]} = 1111 \)
- Repeatedly apply equations
  - \( \text{IN[b]} = \text{OUT[b1]} \cap \ldots \cap \text{OUT[bn]} \)
  - \( \text{OUT[b]} = (\text{IN[b]} - \text{KILL[b]}) \cup \text{GEN[b]} \)
- Use a worklist algorithm to reach fixed point
Available Expressions Algorithm

for all nodes n in N
    OUT[n] = E; // OUT[n] = E - KILL[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] = E; // E is set of all expressions
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ 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 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?
General Correctness

• Concept in actual program execution
  – Reaching definition: definition D, execution E at program point P
  – Available expression: expression X, execution E at program point P
• Analysis reasons about all possible executions
• For all executions E at program point P,
  – if a definition D reaches P in E
  – then D is in the set of reaching definitions at P from analysis
• Other way around
  – if D is not in the set of reaching definitions at P from analysis
  – then D never reaches P in any execution E
• For all executions E at program point P,
  – if an expression X is in set of available expressions at P from analysis
  – then X is available in E at P
• Concept of being conservative
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

\[
\begin{align*}
\text{a} &= \text{x} + \text{y}; \\
\text{t} &= \text{a}; \\
\text{c} &= \text{a} + \text{x}; \\
\text{x} &= \text{0}
\end{align*}
\]

\[
\begin{align*}
\text{b} &= \text{t} + \text{z}; \\
\text{c} &= \text{y} + \text{1}; \\
\text{a} &= \text{1000111} \\
\text{b} &= \text{1000111} \\
\text{c} &= \text{1100100} \\
\text{x} &= \text{1100000} \\
\text{y} &= \text{1110000} \\
\text{z} &= \text{0101110} \\
\text{t} &= \text{1100111}
\end{align*}
\]
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 }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

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

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U 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] = emptyset;
  IN[Entry] = emptyset;
  OUT[Entry] = GEN[Entry];
  Changed = N - { Entry };
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 };
Available Expressions
for all nodes n in N
  OUT[n] = E;
  IN[Entry] = emptyset;
  OUT[Entry] = GEN[Entry];
  Changed = N - { Entry };
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 };
Liveness
for all nodes n in N - { Exit }
  IN[n] = emptyset;
  OUT[Exit] = emptyset;
  IN[Exit] = use[Exit];
  Changed = N - { Exit };
while (Changed != emptyset)
  choose a node n in Changed;
  Changed = Changed - { n };
  OUT[n] = emptyset;
  for all nodes s in successors(n)
    OUT[n] = OUT[n] U 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 };

Saman Amarasinghe
## Comparison

### Reaching Definitions

<table>
<thead>
<tr>
<th>for all nodes n in N</th>
<th>Available Expressions for all nodes n in N</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT[n] = emptyset;</td>
<td>OUT[n] = E;</td>
</tr>
<tr>
<td>IN[Entry] = emptyset;</td>
<td>IN[Entry] = emptyset;</td>
</tr>
<tr>
<td>OUT[Entry] = GEN[Entry];</td>
<td>OUT[Entry] = GEN[Entry];</td>
</tr>
<tr>
<td>Changed = N - { Entry };</td>
<td>Changed = N - { Entry };</td>
</tr>
<tr>
<td>while (Changed != emptyset)</td>
<td>while (Changed != emptyset)</td>
</tr>
<tr>
<td>choose a node n in Changed;</td>
<td>choose a node n in Changed;</td>
</tr>
<tr>
<td>Changed = Changed - { n };</td>
<td>Changed = Changed - { n };</td>
</tr>
</tbody>
</table>

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

| if (OUT[n] changed)                 | if (OUT[n] changed)                        |
|     for all nodes s in successors(n) |     for all nodes s in successors(n)       |
|     Changed = Changed U { s };      |     Changed = Changed U { s };             |

---

Saman Amarasinghe

79

6.035 ©MIT Fall 1998
## Comparison

### Reaching Definitions

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{OUT}[n] = \text{emptyset}; )</td>
<td>for all nodes ( n ) in ( N )</td>
</tr>
<tr>
<td>( \text{IN}[\text{Entry}] = \text{emptyset}; )</td>
<td></td>
</tr>
<tr>
<td>( \text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]; )</td>
<td></td>
</tr>
<tr>
<td>( \text{Changed} = N - { \text{Entry} }; )</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

\[
\text{if } (\text{OUT}[n] \text{ changed}) \\
\quad \text{for all nodes } s \text{ in } \text{successors}(n) \\
\quad \text{Changed} = \text{Changed} \cup \{ s \}; \\
\]

### Liveness

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{IN}[n] = \text{emptyset}; )</td>
<td>for all nodes ( n ) in ( N )</td>
</tr>
<tr>
<td>( \text{OUT}[\text{Exit}] = \text{emptyset}; )</td>
<td></td>
</tr>
<tr>
<td>( \text{IN}[\text{Exit}] = \text{use}[\text{Exit}]; )</td>
<td></td>
</tr>
<tr>
<td>( \text{Changed} = N - { \text{Exit} }; )</td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
\text{IN}[n] = \text{use}[n] \cup (\text{OUT}[n] - \text{def}[n]); \\
\]

\[
\text{if } (\text{IN}[n] \text{ changed}) \\
\quad \text{for all nodes } p \text{ in } \text{predecessors}(n) \\
\quad \text{Changed} = \text{Changed} \cup \{ p \}; \\
\]
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