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
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

```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 $s = s + a \times b$?

Yes!

On all reaching definitions

$a = 4$
Constant Propagation Transform

Yes!
On all reaching definitions

\[ a = 4 \]

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

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

\[ i < n \]

\[ s = s + 4 \times b; \]
\[ i = i + 1; \]

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

No!

One reaching definition with \( b = 1 \)

One reaching definition with \( b = 2 \)
Splitting
Preserves Information Lost At Merges

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

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

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

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

\[ return s \]

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

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

b = 1;
b = 2;
i < n
s = s + a*b;
i = i + 1;

return s

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

b = 1;
b = 2;
i < n
s = s + a*1;
i = i + 1;

return s

s = s + a*2;
i = i + 1;

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;
k == 0
4: b = 1;
5: b = 2;
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
- $\text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b]$  
- $\text{IN}[\text{entry}] = 00000000$
- 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
    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;

\[ k == 0 \]

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

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

\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?

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

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

\[ g = a + c \]

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

NO!
Is the Expression Available?

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

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

NO!
Is the Expression Available?

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

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

NO!
Is the Expression Available?

YES!

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

\[ g = a + c \]

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

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

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

\[ g = a + c \]

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

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

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

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

\[ g = a + c \]

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

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

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

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

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

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

\[ b = a + d \]

\[ g = f \]

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

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

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

\[ j = f + b + d \]

\[ g = f \]

\[ b - a + d \]
\[ h = c + f \]
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 \textit{ANY} predecessor in CFG
  - expression is available at a basic block only if it is available from \textit{ALL} predecessors in CFG
Expressions
1: x+y
2: i\textless{}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

\[ a = x+y; \]
\[ t = a \]
\[ x == 0 \]

\[ x = z; \]
\[ b = x+y; \]
\[ t = b \]

\[ i = t; \]

\[ i < n \]

\[ c = t; \]
\[ i = i+c; \]

\[ d = t \]
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 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

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

- Use fixed point algorithm
- \( \text{IN[entry]} = 0000 \)
- Initialize \( \text{OUT[b]} = 1111 \)
- Repeatedly apply equations
  - \( \text{IN[b]} = \text{OUT}[b_1] \cap ... \cap \text{OUT}[b_n] \)
  - \( \text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b] \)
- 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] \cap 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
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 $abcxyzt$

```plaintext
a = x+y;
t = a;
c = a+x;
x == 0

b = t+z;
c = y+1;
```

```
0101110
a = x+y;
t = a;
c = a+x;
x == 0
1100111
abcxyzt
```

```
1000111
b = t+z;
1100100
abcxyzt
```

```
1100100
c = y+1;
1110000
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

• $\text{USE}[x = z; x = x+1;] = \{ z \}$ (x not in USE)
• $\text{DEF}[x = z; x = x+1; y = 1;] = \{x, y\}$
• 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 };

Saman Amarasinghe

6.035  ©MIT Fall 1998
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
  \[ \text{OUT}[n] = \text{emptyset}; \]
  \[ \text{IN}[\text{Entry}] = \text{emptyset}; \]
  \[ \text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]; \]
  Changed = N - \{ \text{Entry} \};

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

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

**Available Expressions**

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

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

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

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

**Liveness**

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

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

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

  \[ \text{IN}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]); \]
  if (IN[n] changed)
    for all nodes p in predecessors(n)
      Changed = Changed \cup \{ p \};
### Reaching Definitions

for all nodes \( n \) in \( N \)

\[
\text{OUT}[n] = \text{emptyset};
\]

\[
\text{IN}[\text{Entry}] = \text{emptyset};
\]

\[
\text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}];
\]

\[
\text{Changed} = N - \{\text{Entry}\};
\]

while (\text{Changed} \neq \text{emptyset})

\[
\text{choose a node } n \text{ in } \text{Changed};
\]

\[
\text{Changed} = \text{Changed} - \{ n \};
\]

\[
\text{IN}[n] = \text{emptyset};
\]

for all nodes \( p \) in 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)

\[
\text{for all nodes } s \text{ in successors}(n)
\]

\[
\text{Changed} = \text{Changed} \cup \{ s \};
\]

### Available Expressions

for all nodes \( n \) in \( N \)

\[
\text{OUT}[n] = \text{E};
\]

\[
\text{IN}[\text{Entry}] = \text{emptyset};
\]

\[
\text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}];
\]

\[
\text{Changed} = N - \{\text{Entry}\};
\]

while (\text{Changed} \neq \text{emptyset})

\[
\text{choose a node } n \text{ in } \text{Changed};
\]

\[
\text{Changed} = \text{Changed} - \{ n \};
\]

\[
\text{IN}[n] = \text{E};
\]

for all nodes \( p \) in predecessors(\( n \))

\[
\text{IN}[n] = \text{IN}[n] \cap \text{OUT}[p];
\]

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

if (\text{OUT}[n] changed)

\[
\text{for all nodes } s \text{ in successors}(n)
\]

\[
\text{Changed} = \text{Changed} \cup \{ s \};
\]
<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes $n$ in $N$</td>
<td>for all nodes $n$ in $N$</td>
</tr>
<tr>
<td>$\text{OUT}[n] = \text{emptyset}$;</td>
<td>$\text{IN}[n] = \text{emptyset}$;</td>
</tr>
<tr>
<td>$\text{IN}[\text{Entry}] = \text{emptyset}$;</td>
<td>$\text{OUT}[\text{Exit}] = \text{emptyset}$;</td>
</tr>
<tr>
<td>$\text{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]$;</td>
<td>$\text{IN}[\text{Exit}] = \text{use}[\text{Exit}]$;</td>
</tr>
<tr>
<td>$\text{Changed} = N - { \text{Entry} }$;</td>
<td>$\text{Changed} = N - { \text{Exit} }$;</td>
</tr>
<tr>
<td>while (Changed != emptyset)</td>
<td>while (Changed != emptyset)</td>
</tr>
<tr>
<td>\hspace{1em} choose a node $n$ in Changed;</td>
<td>\hspace{1em} choose a node $n$ in Changed;</td>
</tr>
<tr>
<td>\hspace{1em} Changed $\leftarrow$ Changed $\setminus { n }$;</td>
<td>\hspace{1em} Changed $\leftarrow$ Changed $\setminus { n }$;</td>
</tr>
<tr>
<td>$\text{IN}[n] = \text{emptyset}$;</td>
<td>$\text{OUT}[n] = \text{emptyset}$;</td>
</tr>
<tr>
<td>\hspace{1em} for all nodes $p$ in predecessors($n$)</td>
<td>\hspace{1em} for all nodes $s$ in successors($n$)</td>
</tr>
<tr>
<td>\hspace{2em} $\text{IN}[n] = \text{IN}[n] \cup \text{OUT}[p]$;</td>
<td>\hspace{2em} $\text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[p]$;</td>
</tr>
<tr>
<td>$\text{OUT}[n] = \text{GEN}[n] \cup (\text{IN}[n] - \text{KILL}[n])$;</td>
<td>$\text{IN}[n] = \text{use}[n] \cup (\text{OUT}[n] - \text{def}[n])$;</td>
</tr>
<tr>
<td>if (OUT[$n$] changed)</td>
<td>if (IN[$n$] changed)</td>
</tr>
<tr>
<td>\hspace{1em} for all nodes $s$ in successors($n$)</td>
<td>\hspace{1em} for all nodes $p$ in predecessors($n$)</td>
</tr>
<tr>
<td>\hspace{2em} Changed $\leftarrow$ Changed $\cup { s }$;</td>
<td>\hspace{2em} Changed $\leftarrow$ Changed $\cup { p }$;</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
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.