CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
DATAFLOW ANALYSIS

The slides adapted from Saman Amarasinghe, Martin Rinard and Vikram Adve
Why Dataflow Analysis?

Answers key questions about the flow of values and other program properties over control-flow paths at compile-time
Why Dataflow Analysis?

**Compiler fundamentals**
What defs. of \( x \) reach a given use of \( x \) (and vice-versa)?:
What \( \{<\text{ptr},\text{target}>\} \) pairs are possible at each statement?

**Scalar dataflow optimizations**
Are any uses reached by a particular definition of \( x \)?
Has an expression been computed on all incoming paths?
What is the innermost loop level at which a variable is defined?

**Correctness and safety:**
Is variable \( x \) defined on every path to a use of \( x \)?
Is a pointer to a local variable live on exit from a procedure?

**Parallel program optimization**
Where is dataflow analysis used?

Everywhere
Where is dataflow analysis used?

**Preliminary Analyses**
- Pointer Analysis
- Detecting uninitialized variables
- Type inference
- Strength Reduction for Induction Variables

**Static Computation Elimination**
- Dead Code Elimination (DCE)
- Constant Propagation
- Copy Propagation

**Redundancy Elimination**
- Local Common Subexpression Elimination (CSE)
- Global Common Subexpression Elimination (GCSE)
- Loop-invariant Code Motion (LICM)
- Partial Redundancy Elimination (PRE)

**Code Generation**
- Liveness analysis for register allocation
Basic Term Review

**Point:** A location in a basic block just before or after some statement.

**Path:** A path from points p1 to pn is a sequence of points p1, p2, ... pn such that (intuitively) some execution can visit these points in order.

**Kill of a Definition:** A definition d of variable V is killed on a path if there is an unambiguous (re)definition of V on that path.

**Kill of an Expression:** An expression e is killed on a path if there is a possible definition of any of the variables of e on that path.
Dataflow Analysis (Informally)

Symbolically simulate execution of program
• Forward (Reaching Definitions)
• Backward (Variable Liveness)

Stacked analyses and transformations that work together, e.g.
• Reaching Definitions $\rightarrow$ Constant Propagation
• Variable Liveness $\rightarrow$ Dead code elimination

Our plan:
• Examples first (analysis + theory)
• Theory follows
Analysis: Reaching Definitions

A definition $d$ reaches point $p$ if there is a path from the point after $d$ to $p$ such that $d$ is not killed along that path.

Example Statements:

- $a = x + y$
- It is a definition of $a$
- It is a use of $x$ and $y$
- $b = a + 1$
- It is a definition of $?$. And use of $??$

A definition reaches a use if the value written by the definition may be read by the use.
Reaching Definitions

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

\[
b = 1; \\
b = 2; \\
i < n \\
s = s + a\times b; \\
i = i + 1; \\
return s
\]
Reaching Definitions

\[ s_0 = 0; \]
\[ a_0 = 4; \]
\[ i_0 = 0; \]
\[ k_0 = 0 \]
\[ b_0 = 1; \]
\[ b_1 = 2; \]
\[ i < n_0 \]

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

\[ \varphi(b_0, b_1) \]
\[ \varphi(i_0, i_2) \]
\[ \varphi(s_0, s_2) \]

\[ \text{return } s \]
Reaching Definitions

\[
\begin{align*}
    s0 &= 0; \\
    a0 &= 4; \\
    i0 &= 0; \\
    k0 &= 0 \\
    b0 &= 1; \\
    b1 &= 2; \\
    \text{i1} &= \text{i0} + 1; \\
    \text{i2} &= \text{i1} + 1; \\
    \text{s2} &= \text{s1} + a0 \times b2; \\
    \text{s1} &= \phi(s0, s2); \\
    \text{b2} &= \phi(b0, b1); \\
    \text{i1} &= \phi(i0, i2); \\
    \text{return} s \\
\end{align*}
\]
**Transform: Constant Propagation**

Paired with reaching definitions (uses its results)

**Check:** 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 Being Constant in $s = s + a \times b$?

Yes!

$a = 4$
Is a Being Constant in $s = s + a \cdot b$?

Yes!

$s = 0$
$a = 4$
$i = 0$
$k == 0$

$b = 1$

$b = 2$

$i < n$

$s = s + 4 \cdot b$
$i = i + 1$

return $s$

$a = 4$
Reaching Definitions

Dataflow variables (for each block B)

\( \text{In}(B) \equiv \) the set of defs that reach the point before first statement in B

\( \text{Out}(B) \equiv \) the set of defs that reach the point after last statement in B

\( \text{Gen}(B) \equiv \) the set of defs in B that are not killed in B.

\( \text{Kill}(B) \equiv \) the set of all defs that are killed in B (i.e., on the path from entry to exit of B, if def \( d \) \( \notin \) B; or on the path from \( d \) to exit of B, if def \( d \in \) B).

*The difference:*

\( \text{In}(B), \text{Out}(B) \) are **global** dataflow properties.

\( \text{Gen}(B), \text{Kill}(B) \) are **local** properties of block B alone.
Computing Reaching Definitions

Compute with sets of definitions
  • represent sets using bit vectors data structure
  • 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

Perform computation by simulating execution of program until reach fixed point
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1110000
4: b = 1;
  1111000
5: b = 2;
  1110100
6: s = s + a*b;
  return s
7: i = i + 1;
  0101111
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

Example:

• **GEN**[6: s = s + a*b; 7: i = i + 1;] = 0000011
• **KILL**[6: s = s + a*b; 7: i = i + 1;] = 1010000

Compiler scans each basic block to derive GEN and KILL sets
Dataflow Equations

IN and OUT combine the properties from the neighboring blocks in CFG

\[
IN[b] = OUT[b_1] \cup \ldots \cup OUT[bn]
\]
- where \( b_1, \ldots, bn \) are predecessors of \( b \) in CFG

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

\[
IN[\text{entry}] = 0000000
\]

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
Order of the Analysis?

**Goal:** Propagate information as far as possible in each iteration

**Random** – Select the next node randomly

**Preorder** – Select the next node, than explore children in depth-first fashion

**Postorder** – Before selecting the node, explore all its children

**Reverse Postorder** – Explore the node, than explore all its children
  - Opposite from postorder
  - Not the same as preorder!
Reaching Definitions Algorithm

for all nodes $n$ in $N$
  \[
  \text{OUT}[n] = \text{emptyset}; \quad // \quad \text{OUT}[n] = \text{GEN}[n];
  \]

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

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

$\text{Changed} = N - \{ \text{Entry} \}; \quad // \quad N = \text{all nodes in graph}$

while (\text{Changed} \neq \text{emptyset})
  choose a node $n$ in 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 $\text{successors}(n)$
    $\text{Changed} = \text{Changed} \cup \{ s \};$
Reaching Definitions: Convergence

Out[B] is finite
Out[B] never decreases for any B
⇒ must eventually stop changing
At most n iterations if n blocks
⇐ Definitions need propagate only over acyclic paths
Analysis: Available Expressions

An expression $x+y$ is available at a point $p$ if
1. Every path from the initial node to $p$ must evaluate $x+y$ before reaching $p$,
2. There are no assignments to $x$ or $y$ after the expression 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
• Beyond SSA-form analyses
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!**
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?  

**NO!**

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

\[j = a + b + c + d\]
Is the Expression Available? NO!

\[ a = b + c \]
\[ d = e + f \]
\[ f = a + c \]
\[ g = a + c \]
\[ b = a + d \]
\[ h = c + f \]
\[ j = a + b + c + d \]
Transformation: Common Subexpression Elimination

Uses the results of available expressions

Check:

• If the expression is available and computed before,

Transform:

• At the first location, create a temporary variable
• Replace the latter occurrence(s) with the temporary variable name.
Use of Available Expression

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

YES!
Use of Available Expression

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

YES!
Use of Available Expression

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

\[ g = f' \]

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

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

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

\[ g = f' \]

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

\[ j = f' + b + d \]
**Compute 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
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

• Compiler scans each basic block to derive **GEN** and **KILL** sets
Dataflow Equations

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

- Use fixed point algorithm
- \(\text{IN}[\text{entry}] = 0000\)
- Initialize \(\text{OUT}[b] = 1111\)
- Repeatedly apply equations
  - \(\text{IN}[b] = \text{OUT}[b1] \cup ... \cup \text{OUT}[bn]\)
  - \(\text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b]\)
- 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] \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?
Analysis: Variable Liveness

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 they are live on exit
- Assume $x,y,z,t$ not visible outside method
- Represent Liveness Using Bit Vector
  - order is $abcxyzt$
Transformation: Dead Code Elimination

- Assume a,b,c visible outside method
  - So they are live on exit
- Assume x,y,z,t not visible outside method
- Represent Liveness Using Bit Vector
  - order is abcxyzt
- Remove dead definitions
Transformation: Dead Code Elimination

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

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

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

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

  \[\text{IN}[n] = \text{use}[n] U (\text{out}[n] - \text{def}[n]);\]

  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
<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes ( n ) in ( N )</td>
<td>for all nodes ( n ) in ( N )</td>
<td>for all nodes ( n ) in ( N - { \text{Exit} } )</td>
</tr>
<tr>
<td>( \text{OUT}[n] = \text{emptyset}; )</td>
<td>( \text{OUT}[n] = E; )</td>
<td>( \text{IN}[n] = \text{emptyset}; )</td>
</tr>
<tr>
<td>( \text{IN}[\text{Entry}] = \text{emptyset}; )</td>
<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{OUT}[\text{Entry}] = \text{GEN}[\text{Entry}]; )</td>
<td>( \text{OUT}[\text{Exit}] = \text{emptyset}; )</td>
</tr>
<tr>
<td>( \text{Changed} = N - { \text{Entry} }; )</td>
<td>( \text{Changed} = N - { \text{Entry} }; )</td>
<td>( \text{Changed} = N - { \text{Exit} }; )</td>
</tr>
<tr>
<td>while (( \text{Changed} \neq \text{emptyset} ))</td>
<td>while (( \text{Changed} \neq \text{emptyset} ))</td>
<td>while (( \text{Changed} \neq \text{emptyset} ))</td>
</tr>
<tr>
<td>choose a node ( n ) in ( \text{Changed} );</td>
<td>choose a node ( n ) in ( \text{Changed} );</td>
<td>choose a node ( n ) in ( \text{Changed} );</td>
</tr>
<tr>
<td>( \text{Changed} = \text{Changed} - { n }; )</td>
<td>( \text{Changed} = \text{Changed} - { n }; )</td>
<td>( \text{Changed} = \text{Changed} - { n }; )</td>
</tr>
<tr>
<td>( \text{IN}[n] = \text{emptyset}; )</td>
<td>( \text{IN}[n] = E; )</td>
<td>( \text{OUT}[n] = \text{emptyset}; )</td>
</tr>
<tr>
<td>for all nodes ( p ) in ( \text{predecessors}(n) )</td>
<td>for all nodes ( p ) in ( \text{predecessors}(n) )</td>
<td>for all nodes ( s ) in ( \text{successors}(n) )</td>
</tr>
<tr>
<td>( \text{IN}[n] = \text{IN}[n] \cup \text{OUT}[p]; )</td>
<td>( \text{IN}[n] = \text{IN}[n] \cap \text{OUT}[p]; )</td>
<td>( \text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[p]; )</td>
</tr>
<tr>
<td>( \text{OUT}[n] = \text{GEN}[n] \cup (\text{IN}[n] \cap \text{KILL}[n]); )</td>
<td>( \text{OUT}[n] = \text{GEN}[n] \cup (\text{IN}[n] \cap \text{KILL}[n]); )</td>
<td>( \text{IN}[n] = \text{use}[n] \cup (\text{OUT}[n] \cap \text{def}[n]); )</td>
</tr>
<tr>
<td>if (( \text{OUT}[n] ) changed)</td>
<td>if (( \text{OUT}[n] ) changed)</td>
<td>if (( \text{IN}[n] ) changed)</td>
</tr>
<tr>
<td>for all nodes ( s ) in ( \text{successors}(n) )</td>
<td>for all nodes ( s ) in ( \text{successors}(n) )</td>
<td>for all nodes ( p ) in ( \text{predecessors}(n) )</td>
</tr>
<tr>
<td>( \text{Changed} = \text{Changed} \cup { s }; )</td>
<td>( \text{Changed} = \text{Changed} \cup { s }; )</td>
<td>( \text{Changed} = \text{Changed} \cup { p }; )</td>
</tr>
</tbody>
</table>
## Comparison

### Reaching Definitions

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

### Available Expressions

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

### Reaching Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes n in N</td>
<td>for all nodes n in N</td>
</tr>
<tr>
<td>OUT[n] = emptyset;</td>
<td>IN[n] = emptyset;</td>
</tr>
<tr>
<td>IN[Entry] = emptyset;</td>
<td>OUT[Exit] = emptyset;</td>
</tr>
<tr>
<td>OUT[Entry] = GEN[Entry];</td>
<td>IN[Exit] = use[Exit];</td>
</tr>
<tr>
<td>Changed = N - { Entry };</td>
<td>Changed = N - { Exit };</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[n] = emptyset;</td>
<td>OUT[n] = emptyset;</td>
</tr>
<tr>
<td>for all nodes p in predecessors(n)</td>
<td>for all nodes s in successors(n)</td>
</tr>
<tr>
<td>IN[n] = IN[n] U OUT[p];</td>
<td>OUT[n] = OUT[n] U IN[p];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
</table>

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

### Liveness

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all nodes n in N</td>
<td>for all nodes n in N</td>
</tr>
<tr>
<td>IN[n] = emptyset;</td>
<td>OUT[n] = emptyset;</td>
</tr>
<tr>
<td>OUT[Entry] = GEN[Entry];</td>
<td>IN[Exit] = use[Exit];</td>
</tr>
<tr>
<td>Changed = N - { Entry };</td>
<td>Changed = N - { Exit };</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[n] = emptyset;</td>
<td>OUT[n] = emptyset;</td>
</tr>
<tr>
<td>for all nodes p in predecessors(n)</td>
<td>for all nodes s in successors(n)</td>
</tr>
<tr>
<td>IN[n] = IN[n] U OUT[p];</td>
<td>OUT[n] = OUT[n] U IN[p];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Code</th>
</tr>
</thead>
</table>

if (IN[n] changed)
- for all nodes p in predecessors(n)
  Changed = Changed U { p };
WHY?
Basic Idea

Information about program represented using values from algebraic structure called **lattice**

Analysis produces lattice value for each program point

**Two flavors** of analysis

- Forward dataflow analysis [e.g., Reachability]
- Backward dataflow analysis [e.g. Live Variables]
Partial Orders

Set $P$

Partial order relation $\leq$ such that $\forall x, y, z \in P$

- $x \leq x$ (reflexive)
- $x \leq y$ and $y \leq x$ implies $x = y$ (asymmetric)
- $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)

Can use partial order to define

- Upper and lower bounds
- Least upper bound
- Greatest lower bound
Upper Bounds

If $S \subseteq P$ then

• $x \in P$ is an upper bound of $S$ if $\forall y \in S. y \leq x$
• $x \in P$ is the least upper bound of $S$ if
  • $x$ is an upper bound of $S$, and
  • $x \leq y$ for all upper bounds $y$ of $S$
• $\lor$ - $\text{join}$, least upper bound, $\text{lub}$, supremum, $\text{sup}$
  • $\lor S$ is the least upper bound of $S$
  • $x \lor y$ is the least upper bound of $\{x,y\}$
Lower Bounds

If \( S \subseteq P \) then

• \( x \in P \) is a lower bound of \( S \) if \( \forall y \in S. \ x \leq y \)

• \( x \in P \) is the greatest lower bound of \( S \) if
  • \( x \) is a lower bound of \( S \), and
  • \( y \leq x \) for all lower bounds \( y \) of \( S \)

• \( \wedge \) - meet, greatest lower bound, glb, infimum, \( \inf \)
  • \( \wedge S \) is the greatest lower bound of \( S \)
  • \( x \wedge y \) is the greatest lower bound of \( \{ x, y \} \)
Covering

$x < y$ if $x \leq y$ and $x \neq y$

**x is covered by y** (y covers x) if

- $x < y$, and
- $x \leq z < y$ implies $x = z$

Conceptually, y covers x if there are no elements between x and y
Example

\[ P = \{ \text{000}, \text{001}, \text{010}, \text{011}, \text{100}, \text{101}, \text{110}, \text{111} \} \]

(standard boolean lattice, also called hypercube)

\[ x \leq y \text{ if } (x \text{ bitwise and } y) = x \]

Hasse Diagram

- If \( y \) covers \( x \)
  - Line from \( y \) to \( x \)
  - \( y \) above \( x \) in diagram
Lattices

If \( x \land y \) and \( x \lor y \) exist for all \( x, y \in P \),
then \( P \) is a lattice.

If \( \land S \) and \( \lor S \) exist for all \( S \subseteq P \),
then \( P \) is a complete lattice.

All finite lattices are complete

Example of a lattice that is not complete

- Integers \( I \)
- For any \( x, y \in I \), \( x \lor y = \max(x, y) \), \( x \land y = \min(x, y) \)
- But \( \lor I \) and \( \land I \) do not exist
- \( I \cup \{+\infty, -\infty\} \) is a complete lattice
Top and Bottom

Greatest element of $P$ (if it exists) is top ($\top$)
Least element of $P$ (if it exists) is bottom ($\bot$)
Connection Between $\leq$, $\land$, and $\lor$

The following 3 properties are equivalent:

- $x \leq y$
- $x \lor y = y$
- $x \land y = x$

Will prove:

- $x \leq y$ implies $x \lor y = y$ and $x \land y = x$
- $x \lor y = y$ implies $x \leq y$
- $x \land y = x$ implies $x \leq y$

Then by transitivity, can obtain

- $x \lor y = y$ implies $x \land y = x$
- $x \land y = x$ implies $x \lor y = y$
Connecting Lemma Proofs

Prove: $x \leq y$ implies $x \lor y = y$

- $x \leq y$ implies $y$ is an upper bound of $\{x,y\}$.
- Any upper bound $z$ of $\{x,y\}$ must satisfy $y \leq z$.
- So $y$ is least upper bound of $\{x,y\}$ and $x \lor y = y$

Prove: $x \leq y$ implies $x \land y = x$

- $x \leq y$ implies $x$ is a lower bound of $\{x,y\}$.
- Any lower bound $z$ of $\{x,y\}$ must satisfy $z \leq x$.
- So $x$ is greatest lower bound of $\{x,y\}$ and $x \land y = x$
Connecting Lemma Proofs

Prove: $x \lor y = y$ implies $x \leq y$
- $y$ is an upper bound of $\{x, y\}$ implies $x \leq y$

Prove: $x \land y = x$ implies $x \leq y$
- $x$ is a lower bound of $\{x, y\}$ implies $x \leq y$
Lattices as Algebraic Structures

We have defined \( \lor \) and \( \land \) in terms of \( \leq \).

We will now define \( \leq \) in terms of \( \lor \) and \( \land \):

- Start with \( \lor \) and \( \land \) as arbitrary algebraic operations that satisfy associative, commutative, idempotence, and absorption laws.
- Will define \( \leq \) using \( \lor \) and \( \land \).
- Will show that \( \leq \) is a partial order.

Intuitive concept of \( \lor \) and \( \land \) as information combination operators (or, and).
Algebraic Properties of Lattices

Assume arbitrary operations $\lor$ and $\land$ such that

- $(x \lor y) \lor z = x \lor (y \lor z)$ (associativity of $\lor$)
- $(x \land y) \land z = x \land (y \land z)$ (associativity of $\land$)
- $x \lor y = y \lor x$ (commutativity of $\lor$)
- $x \land y = y \land x$ (commutativity of $\land$)
- $x \lor x = x$ (idempotence of $\lor$)
- $x \land x = x$ (idempotence of $\land$)
- $x \lor (x \land y) = x$ (absorption of $\lor$ over $\land$)
- $x \land (x \lor y) = x$ (absorption of $\land$ over $\lor$)
Connection Between $\land$ and $\lor$

$x \lor y = y$ if and only if $x \land y = x$

Proof of $x \lor y = y$ implies $x = x \land y$

$$x = x \land (x \lor y) \quad \text{(by absorption)}$$
$$= x \land y \quad \text{(by assumption)}$$

Proof of $x \land y = x$ implies $y = x \lor y$

$$y = y \lor (y \land x) \quad \text{(by absorption)}$$
$$= y \lor (x \land y) \quad \text{(by commutativity)}$$
$$= y \lor x \quad \text{(by assumption)}$$
$$= x \lor y \quad \text{(by commutativity)}$$
Properties of $\leq$

Define $x \leq y$ if $x \lor y = y$

Proof of transitive property. Must show that $x \lor y = y$ and $y \lor z = z$ implies $x \lor z = z$

\[
x \lor z = x \lor (y \lor z) \quad \text{(by assumption)}
\]
\[
= (x \lor y) \lor z \quad \text{(by associativity)}
\]
\[
= y \lor z \quad \text{(by assumption)}
\]
\[
= z \quad \text{(by assumption)}
\]
Properties of $\leq$

Proof of asymmetry property. Must show that $x \lor y = y$ and $y \lor x = x$ implies $x = y$

$$x = y \lor x \quad \text{(by assumption)}$$

$$= x \lor y \quad \text{(by commutativity)}$$

$$= y \quad \text{(by assumption)}$$

Proof of reflexivity property. Must show that $x \lor x = x$

$$x \lor x = x \quad \text{(by idempotence)}$$
Properties of $\leq$

Induced operation $\leq$ agrees with original definitions of $\vee$ and $\wedge$, i.e.,

- $x \vee y = \sup \{x, y\}$
- $x \wedge y = \inf \{x, y\}$
Proof of $x \lor y = \sup \{x, y\}$

Consider any upper bound $u$ for $x$ and $y$. Given $x \lor u = u$ and $y \lor u = u$, must show $x \lor y \leq u$, i.e., $(x \lor y) \lor u = u$

$$u = x \lor u \quad \text{(by assumption)}$$
$$= x \lor (y \lor u) \quad \text{(by assumption)}$$
$$= (x \lor y) \lor u \quad \text{(by associativity)}$$
Proof of $x \wedge y = \inf \{x, y\}$

• Consider any lower bound $l$ for $x$ and $y$.
• Given $x \wedge l = l$ and $y \wedge l = l$, must show $l \leq x \wedge y$, i.e., $(x \wedge y) \wedge l = l$

\[
\begin{align*}
l &= x \wedge l && \text{(by assumption)} \\
&= x \wedge (y \wedge l) && \text{(by assumption)} \\
&= (x \wedge y) \wedge l && \text{(by associativity)}
\end{align*}
\]
Chains

A set $S$ is a chain if $\forall x, y \in S. y \leq x$ or $x \leq y$

$P$ has no infinite chains if every chain in $P$ is finite

$P$ satisfies the ascending chain condition if for all sequences $x_1 \leq x_2 \leq \ldots$ there exists $n$ such that $x_n = x_{n+1} = \ldots$
Application to Dataflow Analysis

Dataflow information will be lattice values

• **Transfer functions** operate on lattice values
• Solution algorithm will generate *increasing sequence of values* at each program point
• Ascending chain condition will ensure **termination**

We will use $\lor$ to combine values at control-flow join points
Transfer Functions

Transfer function $f : P \rightarrow P$ for each node in control flow graph

$f$ models effect of the node on the program information
Transfer Functions

Each dataflow analysis problem has a set $F$ of transfer functions $f: P \rightarrow P$, i.e.,

- **Identity function** belongs to the set, $i \in F$
- $F$ must be **closed under composition**:
  $$\forall f,g \in F. \text{ the function } h = \lambda x. f(g(x)) \in F$$
- Each $f \in F$ must be **monotone**:
  $$x \leq y \text{ implies } f(x) \leq f(y)$$
- Sometimes all $f \in F$ are **distributive**:
  $$f(x \lor y) = f(x) \lor f(y)$$
- Note that Distributivity implies monotonicity
Distributivity Implies Monotonicity

Proof.
Assume distributivity: \( f(x \lor y) = f(x) \lor f(y) \)

Must show: \( x \lor y = y \) implies \( f(x) \lor f(y) = f(y) \)

\[
\begin{align*}
    f(y) &= f(x \lor y) \quad \text{(by assumption)} \\
    &= f(x) \lor f(y) \quad \text{(by distributivity)}
\end{align*}
\]
Putting Pieces Together

Forward Dataflow Analysis Framework
Simulates execution of program forward with flow of control
Forward Dataflow Analysis

Simulates execution of program forward with flow of control

For each node $n$, have

- $in_n$ – value at program point before $n$
- $out_n$ – value at program point after $n$
- $f_n$ – transfer function for $n$ (given $in_n$, computes $out_n$)

Require that solution satisfy

- $\forall n. \ out_n = f_n(in_n)$
- $\forall n \neq n_0. \ in_n = \lor \{ \ out_m \ . \ m \ in \ pred(n) \}$
- $in_{n_0} = I$
- Where $I$ summarizes information at start of program
Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

\[
\text{out}_n := f_n(\text{in}_n) \\
\text{in}_n := \lor \{ \text{out}_m \cdot m \text{ in pred}(n) \}
\]

Conceptually separates analysis problem from program
Worklist Algorithm for Solving Forward Dataflow Equations

for each n do out\n := f_n(⊥)

in\n := I; out\n := f_n(1)

worklist := N - \{ n_0 \}

while worklist ≠ ∅ do
  remove a node n from worklist
  in\n := ∨ \{ out_m . m in pred(n) \}
  out\n := f_n(in\n)
  if out\n changed then
    worklist := worklist ∪ succ(n)
Correctness Argument

Why does result satisfy dataflow equations?

Whenever process a node $n$, algorithm sets $\text{out}_n := f_n(\text{in}_n)$
Algorithm ensures that $\text{out}_n = f_n(\text{in}_n)$
Whenever $\text{out}_m$ changes, put $\text{succ}(m)$ on worklist.
Consider any node $n \in \text{succ}(m)$. It will eventually come off worklist and algorithm will set

$$\text{in}_n := \vee \{ \text{out}_m . m \in \text{pred}(n) \}$$

to ensure that $\text{in}_n = \vee \{ \text{out}_m . m \in \text{pred}(n) \}$
So final solution will satisfy dataflow equations
Termination Argument

Why does algorithm terminate?

Sequence of values taken on by $\text{in}_n$ or $\text{out}_n$ is a chain. If values stop increasing, worklist empties and algorithm terminates.

If lattice has ascending chain property, algorithm terminates

- Algorithm terminates for finite lattices
- For lattices without ascending chain property, use widening operator
Widening Operators

Detect lattice values that may be part of infinitely ascending chain
Artificially raise value to least upper bound of chain

Example:

• Lattice is set of all subsets of integers
• Could be used to collect possible values taken on by variable during execution of program
• Widening operator might raise all sets of size n or greater to TOP (likely to be useful for loops)
Reaching Definitions Algorithm

**Reminder**

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

\[
\text{OUT}\[n\] = \text{emptyset}; \quad // \quad \text{OUT}[n] = \text{GEN}[n];
\]

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

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

\[
\text{Changed} = \text{N} - \{ \text{Entry} \}; \quad // \quad \text{N} = \text{all nodes in graph}
\]

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 \( \text{successors}(n) \)

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

\[ P = \text{powerset of set of all definitions in program (all subsets of set of definitions in program)} \]
\[ \forall = \cup \ (\text{order is } \subseteq) \]
\[ \bot = \emptyset \]
\[ l = \text{in}_{n_0} = \bot \]
\[ F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b) \]
  - b is set of definitions that node kills
  - a is set of definitions that node generates

General pattern for many transfer functions
  - f(x) = GEN \cup (x-KILL)
Does Reaching Definitions Framework Satisfy Properties?

\( \subseteq \) satisfies conditions for \( \leq \)

- \( x \subseteq y \) and \( y \subseteq z \) implies \( x \subseteq z \) (transitivity)
- \( x \subseteq y \) and \( y \subseteq x \) implies \( y = x \) (asymmetry)
- \( x \subseteq x \) (idempotence)

\( F \) satisfies transfer function conditions

- \( \lambda x. \emptyset \cup (x - \emptyset) = \lambda x. x \in F \) (identity)
- \textbf{Will show} \( f(x \cup y) = f(x) \cup f(y) \) (distributivity)
  
  \[
  f(x) \cup f(y) = (a \cup (x - b)) \cup (a \cup (y - b)) = a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) = f(x \cup y)
  \]
Does Reaching Definitions Framework Satisfy Properties?

What about composition?

• Given \( f_1(x) = a_1 \cup (x - b_1) \) and \( f_2(x) = a_2 \cup (x - b_2) \)

• Must show \( f_1(f_2(x)) \) can be expressed as \( a \cup (x - b) \)

\[
\begin{align*}
f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x - b_2)) - b_1) \\
&= a_1 \cup ((a_2 - b_1) \cup ((x - b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup ((x - b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup (x - (b_2 \cup b_1))
\end{align*}
\]

• Let \( a = (a_1 \cup (a_2 - b_1)) \) and \( b = b_2 \cup b_1 \)

• Then \( f_1(f_2(x)) = a \cup (x - b) \)
General Result

All GEN/KILL transfer function frameworks satisfy

- Identity
- Distributivity
- Composition

Properties
Available Expressions

\[ P = \text{powerset of set of all expressions in program (all subsets of set of expressions)} \]

\[ \forall = \cap \text{ (order is } \supseteq \text{)} \]

\[ \bot = P \]

\[ I = \text{in}_{n_0} = \emptyset \]

\[ F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b) \]

- b is set of expressions that node kills
- a is set of expressions that node generates

Another GEN/KILL analysis
Concept of Conservatism

Reaching definitions use $\cup$ as join

- Optimizations must take into account all definitions that reach along ANY path

Available expressions use $\cap$ as join

- Optimization requires expression to reach along ALL paths

Optimizations must conservatively take all possible executions into account.
Backward Dataflow Analysis

• Simulates execution of program backward against the flow of control
  • For each node $n$, have
    – $in_n$ – value at program point before $n$
    – $out_n$ – value at program point after $n$
    – $f_n$ – transfer function for $n$ (given $out_n$, computes $in_n$)
  • Require that solution satisfies
    – $\forall n. in_n = f_n(out_n)$
    – $\forall n \notin N_{\text{final}}. out_n = \lor \{ in_m \cdot m \in \text{succ}(n) \}$
    – $\forall n \in N_{\text{final}} = out_n = O$
    – Where $O$ summarizes information at end of program
Worklist Algorithm for Solving Backward Dataflow Equations

for each \( n \) do
\[ \text{in}_n := f_n(\perp) \]

for each \( n \in N_{\text{final}} \) do
\[ \text{out}_n := O; \text{in}_n := f_n(O) \]

\( \text{worklist} := N - N_{\text{final}} \)

while worklist \( \neq \emptyset \) do

remove a node \( n \) from worklist

\[ \text{out}_n := \lor \{ \text{in}_m . m \in \text{succ}(n) \} \]

\[ \text{in}_n := f_n(\text{out}_n) \]

if \( \text{in}_n \) changed then

\[ \text{worklist} := \text{worklist} \cup \text{pred}(n) \]
Live Variables

\[ P = \text{powerset of set of all variables in program} \]
\[ \vee = \cup \ (\text{order is } \subseteq) \]
\[ \bot = \emptyset \]
\[ \mathcal{O} = \emptyset \]

\[ F = \text{all functions } f \text{ of the form } f(x) = a \cup (x - b) \]
  
  - \( b \) is set of variables that node kills
  - \( a \) is set of variables that node reads