CMSC 631
Program Analysis and Understanding
Spring 2013

Data Flow Analysis
Data Flow Analysis

• A framework for proving facts about programs

• Reasons about lots of little facts

• Little or no interaction between facts
  ▪ Works best on properties about how program computes

• Based on all paths through program
  ▪ Including infeasible paths

• Operates on control-flow graphs, typically
\[ x := a + b; \]
\[ y := a \times b; \]
\[ \text{while } (y > a) \{ \]
\[ \quad a := a + 1; \]
\[ \quad x := a + b \]
\[ \}\]
Control-Flow Graph w/Basic Blocks

x := a + b;
y := a * b;
while (y > a + b) {
  a := a + 1;
  x := a + b
}

• Can lead to more efficient implementations
• But more complicated to explain, so...
  ▪ We’ll use single-statement blocks in lecture today
Example with Entry and Exit

x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}

- All nodes without a (normal) predecessor should be pointed to by entry
- All nodes without a successor should point to exit
Notes on Entry and Exit

- Typically, we perform data flow analysis on a function body
- Functions usually have
  - A unique entry point
  - Multiple exit points
- So in practice, there can be multiple exit nodes in the CFG
  - For the rest of these slides, we’ll assume there’s only one
  - In practice, just treat all exit nodes the same way as if there’s only one exit node
Available Expressions

- An expression $e$ is available at program point $p$ if
  - $e$ is computed on every path to $p$, and
  - the value of $e$ has not changed since the last time $e$ was computed on the paths to $p$

- Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it’s still in a register somewhere)
Data Flow Facts

- Is expression e available?
- Facts:
  - $a + b$ is available
  - $a \times b$ is available
  - $a + 1$ is available
Gen and Kill

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := a + b$</td>
<td>$a + b$</td>
<td></td>
</tr>
<tr>
<td>$y := a \times b$</td>
<td>$a \times b$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td></td>
<td>$a + 1$, $a + b$, $a \times b$</td>
</tr>
</tbody>
</table>
Computing Available Expressions

∅

entry

x := a + b

{a + b}

y := a * b

{a + b, a * b}

y > a

{a + b, a * b}

a := a + 1

∅

x := a + b

{a + b}

exit

{a + b}
Terminology

• A joint point is a program point where two branches meet

• Available expressions is a forward must problem
  ▪ Forward = Data flow from in to out
  ▪ Must = At join point, property must hold on all paths that are joined
Data Flow Equations

• Let $s$ be a statement
  - $\text{succ}(s) = \{ \text{immediate successor statements of } s \}$
  - $\text{pred}(s) = \{ \text{immediate predecessor statements of } s \}$
  - $\text{in}(s) = \text{program point just before executing } s$
  - $\text{out}(s) = \text{program point just after executing } s$

• $\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s')$

• $\text{out}(s) = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))$
  - Note: These are also called transfer functions
Liveness Analysis

• A variable $v$ is *live* at program point $p$ if
  - $v$ will be used on some execution path originating from $p$...
  - before $v$ is overwritten

• Optimization
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment
Data Flow Equations

- Available expressions is a forward must analysis
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths

- Liveness is a *backward may* problem
  - To know if variable live, need to look at future uses
  - Variable is live if used on some path

- \[
  \text{out}(s) = \bigcup_{s' \in \text{succ}(s)} \text{in}(s')
\]

- \[
  \text{in}(s) = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s))
\]
# Gen and Kill

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a, b</td>
<td>x</td>
</tr>
<tr>
<td>y := a * b</td>
<td>a, b</td>
<td>y</td>
</tr>
<tr>
<td>y &gt; a</td>
<td>a, y</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>
Computing Live Variables

\{x, y, a, b\}

\{x\}

\{x, y, a, b\}

\{x, y, a, b\}

\{x, y, a, b\}

\{x, y, a, b\}

\{x, y, a, b\}
Very Busy Expressions

- An expression $e$ is *very busy* at point $p$ if
  - On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

- Optimization
  - Can hoist very busy expression computation

- What kind of problem?
  - Forward or backward? *backward*
  - May or must? *must*
A definition of a variable $v$ is an assignment to $v$.

A definition of variable $v$ reaches point $p$ if:
- There is no intervening assignment to $v$.

Also called def-use information.

What kind of problem?
- Forward or backward? forward
- May or must? may
### Space of Data Flow Analyses

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>

- Most data flow analyses can be classified this way
  - A few don’t fit: bidirectional analysis
- Lots of literature on data flow analysis
Solving data flow equations

• Let’s start with forward may analysis
  ■ Dataflow equations:
    - \( \text{in}(s) = \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \)
    - \( \text{out}(s) = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)) \)

• Need algorithm to compute \( \text{in} \) and \( \text{out} \) at each stmt

• Key observation: \( \text{out}(s) \) is monotonic in \( \text{in}(s) \)
  ■ \( \text{gen}(s) \) and \( \text{kill}(s) \) are fixed for a given \( s \)
  ■ If, during our algorithm, \( \text{in}(s) \) grows, then \( \text{out}(s) \) grows
  ■ Furthermore, \( \text{out}(s) \) and \( \text{in}(s) \) have max size

• Same with \( \text{in}(s) \)
  ■ in terms of \( \text{out}(s') \) for predecessors \( s' \)
Solving data flow equations (cont’d)

- Idea: fixpoint algorithm
  - Set \( \text{out(entry)} \) to emptyset
    - E.g., we know no definitions reach the entry of the program
  - Initially, assume \( \text{in(s)}, \text{out(s)} \) empty everywhere else, also
  - Pick a statement \( s \)
    - Compute \( \text{in(s)} \) from predecessors’ \( \text{out’s} \)
    - Compute new \( \text{out(s)} \) for \( s \)
  - Repeat until nothing changes

- Improvement: use a worklist
  - Add statements to worklist if their \( \text{in(s)} \) might change
  - Fixpoint reached when worklist is empty
Forward May Data Flow Algorithm

\[
\begin{align*}
\text{out(entry)} &= \emptyset \\
\text{for all other statements } s: & \\
\text{out(s)} &= \emptyset \\
W &= \text{all statements} \quad \text{ // worklist} \quad \\
\text{while } W \text{ not empty:} & \\
\text{take } s \text{ from } W & \\
\text{in(s)} &= \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \\
\text{temp} &= \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)) \\
\text{if } \text{temp} \neq \text{out(s)} \text{ then} & \\
\text{out(s)} &= \text{temp} \\
W &= W \cup \text{succ}(s) \\
\end{align*}
\]
## Generalizing

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>in(s) = $\bigcup_{s' \in \text{pred}(s)} \text{out}(s')$</td>
<td>in(s) = $\bigcap_{s' \in \text{pred}(s)} \text{out}(s')$</td>
</tr>
<tr>
<td></td>
<td>out(s) = gen(s) $\cup$ (in(s) - kill(s))</td>
<td>out(s) = gen(s) $\cup$ (in(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>out(entry) = $\emptyset$</td>
<td>out(entry) = $\emptyset$</td>
</tr>
<tr>
<td></td>
<td>initial out elsewhere = $\emptyset$</td>
<td>initial out elsewhere = ${\text{all facts}}$</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>out(s) = $\bigcup_{s' \in \text{succ}(s)} \text{in}(s')$</td>
<td>out(s) = $\bigcap_{s' \in \text{succ}(s)} \text{in}(s')$</td>
</tr>
<tr>
<td></td>
<td>in(s) = gen(s) $\cup$ (out(s) - kill(s))</td>
<td>in(s) = gen(s) $\cup$ (out(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>in(exit) = $\emptyset$</td>
<td>in(exit) = $\emptyset$</td>
</tr>
<tr>
<td></td>
<td>initial in elsewhere = $\emptyset$</td>
<td>initial out elsewhere = ${\text{all facts}}$</td>
</tr>
</tbody>
</table>
Outgoing statements analysis:

\[
\text{out(\text{entry})} = \emptyset
\]

for all other statements \( s \)

\[
\text{out}(s) = \emptyset
\]

\[ W = \text{all statements} \quad \text{// worklist} \]

while \( W \) not empty

take \( s \) from \( W \)

\[
\text{in}(s) = \bigcup_{s' \in \text{pred}(s)} \text{out}(s')
\]

\[
\text{temp} = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))
\]

if \( \text{temp} \neq \text{out}(s) \) then

\[
\text{out}(s) = \text{temp}
\]

\[
W := W \cup \text{succ}(s)
\]

end

end

\[
\text{out(\text{entry})} = \emptyset
\]

for all other statements \( s \)

\[
\text{out}(s) = \text{all facts}
\]

\[ W = \text{all statements} \]

while \( W \) not empty

take \( s \) from \( W \)

\[
\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s')
\]

\[
\text{temp} = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))
\]

if \( \text{temp} \neq \text{out}(s) \) then

\[
\text{out}(s) = \text{temp}
\]

\[
W := W \cup \text{succ}(s)
\]

end

end

\[ \text{May} \quad \text{Must} \]
Backward Analysis

in(exit) = ∅
for all other statements s
  in(s) = ∅
W = all statements
while W not empty
  take s from W
    out(s) = \bigcup_{s' \in \text{succ}(s)} \text{in}(s')
    temp = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s))
    if temp ≠ \text{in}(s) then
      in(s) = temp
      W := W \cup \text{pred}(s)
  end
end

May

Must
Practical Implementation

- Represent set of facts as bit vector
  - $\text{Fact}_i$ represented by bit $i$
    - Intersection = bitwise and, union = bitwise or, etc
- “Only” a constant factor speedup
  - But very useful in practice
Generalizing Further

• Observe out(s) is a function of out(s’) for preds s’

\[ \text{out}(s) = \text{gen}(s) \cup \left( \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \right) - \text{kill}(s) \]

- We can define other kinds of functions, to compute other kinds of information using dataflow analysis!

• Example: constant propagation
  - Facts — variable x has value n (at this program point)
  - Not quite gen/kill:

```c
/* facts: a = 1, b = 2 */
x = a + b
/* facts: a = 1, b = 2, x = 3 */
```

- Fact that x is 3 not determined syntactically by statement
- So, how can we use data flow analysis to handle this case?
Partial Orders

- To generalize data flow analysis, need to introduce two mathematical structures:
  - Partial orders
  - Lattices

- A *partial order* (p.o.) is a pair \((P, \leq)\) such that
  - \(\leq \subseteq P \times P\)
  - \(\leq\) is reflexive: \(x \leq x\)
  - \(\leq\) is anti-symmetric: \(x \leq y\) and \(y \leq x\) \(\Rightarrow x = y\)
  - \(\leq\) is transitive: \(x \leq y\) and \(y \leq z\) \(\Rightarrow x \leq z\)
Examples

- $(\mathbb{N}, \leq)$
  - Natural numbers with standard inequality
- $(\mathbb{N} \cup \{\infty\}, \leq)$
  - Natural numbers plus infinity, with standard inequality
- $(\mathbb{Z}, \leq)$
  - Integers with standard inequality
- For any set $S$, $(2^S, \subseteq)$
  - The powerset partial order
- For any set $S$, $(S, =)$
  - The discrete partial order
- A 2-crown $(\{a,b,c,d\}, \{a< c, a< d, b< c, b< d\})$
We can write partial orders as graphs using the following conventions:

- Nodes are elements of the p.o.
- Edge from element lower on page to high on page means lower element is strictly less than higher element.

\[(N, \leq)\]

\[
\begin{array}{cccccc}
\infty & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 1 & 2 & 1 & 0 & -1 \\
\end{array}
\]
Drawing Partial Orders (cont’d)

\[
\begin{align*}
\{a,b,c\}, \subseteq & \quad \{a,b,c\}, = \\
\text{2-crown} &
\end{align*}
\]
Meet and Join Operations

• □ is the meet or greatest lower bound operation:
  - $x \sqcap y \leq x$ and $x \sqcap y \leq y$
  - if $z \leq x$ and $z \leq y$ then $z \leq x \sqcap y$

• ⊔ is the join or least upper bound operation:
  - $x \leq x \sqcup y$ and $y \leq x \sqcup y$
  - if $x \leq z$ and $y \leq z$ then $x \sqcup y \leq z$
Examples

- \((\mathbb{N}, \leq), (\mathbb{N} \cup \{\infty\}, \leq), (\mathbb{Z}, \leq)\)
  - \(\cap = \text{min}, \cup = \text{max}\)
- For any set \(S\), \((2^S, \subseteq)\)
  - \(\cap = \cap, \cup = \cup\)
- For any set \(S\), \((S, =)\)
  - \(\cap\) and \(\cup\) only defined when element is the same
- A 2-crown \(\{(a,b,c,d), \{a < c, a < d, b < c, b < d\}\}\)
  - \(a \cup b\) and \(c \cap d\) undefined
Lattices

• A p.o. is a *lattice* if \( \sqcap \) and \( \sqcup \) are defined on any two elements
  
  ▪ A partial order is a *complete lattice* if \( \sqcap \) and \( \sqcup \) are defined on any set

• A lattice has unique elements \( \bot \) (“bottom”) and \( \top \) (“top”) such that
  
  ▪ \( x \sqcap \bot = \bot \) \hspace{1cm} \( x \sqcup \bot = x \)
  ▪ \( x \sqcap \top = x \) \hspace{1cm} \( x \sqcup \top = \top \)

• In a lattice,
  
  ▪ \( x \leq y \) iff \( x \sqcap y = x \)
  ▪ \( x \leq y \) iff \( x \sqcup y = y \)
Examples

• \((N, \leq)\)
  - \(- = 0, \top \) undefined; is a lattice, but not a complete lattice

• \((N \cup \{\infty\}, \leq)\)
  - \(- = 0, \top = \infty;\) is a complete lattice

• \((Z, \leq)\)
  - \(-, \top \) undefined; is a lattice, but not a complete lattice

• For any set \(S, (2^S, \subseteq)\)
  - \(- = \emptyset, \top = S,\) is a complete lattice

• For any set \(S, (S, =)\)
  - \(-, \top \) undefined; not a lattice

• A 2-crown \(\{a,b,c,d\}, \{a\leq c, a\leq d, b\leq c, b\leq d\}\)
  - \(-, \top \) undefined; not a lattice
Flipping a Lattice

- **Lemma:** If \((P, \leq)\) is a lattice, then \((P, \lambda xy. y \leq x)\) is also a lattice
  - i.e., if we flip the sense of \(\leq\), we still have a lattice

- **Examples:**

\[
\begin{align*}
&\text{0} \\
&\text{1} \\
&\text{2} \\
&\vdots \\
&\vdots \\
&\vdots \\
&\{a, b, c\}, \geq
\end{align*}
\]

\[
\begin{align*}
&\emptyset \\
&a, b \\
&b, c \\
&c, \emptyset \\
&a, b, c \\
&\{a, b, c\}, \geq
\end{align*}
\]
Cross-product lattice

• Lemma: Suppose $(P, \leq_1)$ and $(Q, \leq_2)$ are lattices. Then $(P \times Q, \leq)$ is also a lattice, where
  - $(p,q) \leq (p', q')$ iff $p \leq_1 p'$ and $q \leq_2 q'$
  - (Can also take cross product of more than 2 lattices, in the natural way)

• Examples:

\[
\begin{array}{ccc}
  b & d & (b,d) \\
  \times & & (b,c) \\
  a & c & (a,d)
\end{array}
\]

\[
\begin{array}{ccc}
  (b,d) & (b,c) & (a,d) \\
  (b,c) & (a,d) & (a,c)
\end{array}
\]

\[
\begin{array}{ccc}
  (b,d) & (b,c) & (a,d) \\
  (b,c) & (a,d) & (a,c)
\end{array}
\]

\[
\begin{array}{ccc}
  (b,d,f) & (b,c,f) & (a,d,f) \\
  (b,d,f) & (b,c,f) & (a,d,f)
\end{array}
\]

\[
\begin{array}{ccc}
  (b,d,f) & (b,c,f) & (a,d,f) \\
  (b,d,f) & (b,c,f) & (a,d,f)
\end{array}
\]
Data Flow Facts and Lattices

- Sets of dataflow facts form the powerset lattice
  - Example: Available expressions

```
(a+b, a*b, a+1)
 /     \
(a+b, a*) /   \(a+b, a+1)
 /     \   /     \\
(a*b, a+1)(none)(a*b, a+1)
     \\
(a+b)(a+1)
     \\
(a*b)(none)
```
Transfer Functions

• Recall this step from *forward must* analysis:

\[
\text{in}(s) := \bigcap_{s' \in \text{pred}(s)} \text{out}(s')
\]

\[
\text{temp} := \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))
\]

• Let’s recast this in terms of powerset lattice
  - \(\bigcap\) is \(\cap\) in the lattice
  - \(\text{gen}(s), \text{kill}(s)\) are fixed
    - So \(\text{temp}\) is a function of \(\text{in}(s)\)
  - Putting this together:

\[
\text{in}(s) := \bigcap_{s' \in \text{pred}(s)} \text{out}(s')
\]

\[
\text{temp} := f_s(\text{in}(s))
\]

where \(f_s(x) = \text{gen}(s) \cup (x - \text{kill}(s))\)

\(f_s\) is a *transfer function*
Forward May Analysis

• What about forward may analysis?

\[
\text{in}(s) := \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \\
\text{temp} := \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))
\]

- We can just use the flipped powerset lattice
  - \( \cup \) is \( \cap \) is that lattice
- So we get the same equations

\[
\text{in}(s) := \bigcap_{s' \in \text{pred}(s)} \text{out}(s') \\
\text{temp} := f_s(\text{in}(s)) \\
\text{where } f_s(x) = \text{gen}(s) \cup \text{in}(s) - \text{kill}(s)
\]

• Same idea for must/may backward analysis
  - But still separate from forward analysis
Initial Facts

• Recall also from forward must analysis:

\[ \text{out(entry)} = \emptyset \]
\[ \text{initial out elsewhere} = \{\text{all facts}\} \]

• Values of these in lattice terms depends on analysis
  - Available expressions
    - \( \text{out(entry)} \) is the same as \( \bot \)
    - initial out elsewhere is the same as \( \top \)
  - Reaching definitions (with \( \leq \) as \( \supseteq \))
    - \( \text{out(entry)} \) is \( \emptyset \) which is \( \top \) in this lattice (flipped powerset)
    - initial out elsewhere is also \( \top \)
Data Flow Analysis, over Lattices

out(entry) = (as given)
for all other statements s
   out(s) = ⊤
W = all statements // worklist
while W not empty
   take s from W
   in(s) = ⨍ s' ∈ pred(s) out(s')
   temp = f_s(in(s))
   if temp ≠ out(s) then
      out(s) = temp
      W := W ∪ succ(s)
   end
end

in(exit) = (as given)
for all other statements s
   in(s) = ⊤
W = all statements
while W not empty
   take s from W
   out(s) = ⨍ s' ∈ succ(s) in(s')
   temp = f_s(out(s))
   if temp ≠ in(s) then
      in(s) = temp
      W := W ∪ pred(s)
   end
end

Forward
(Red = varies by analysis)

Backward
DFA over Lattices, cont’d

- A dataflow analysis is defined by 4 things:
  - Forward or backward
  - The lattice
    - Data flow facts
    - \( \cap \) operation
      - In terms of gen/kill dfa, this specifies may or must
    - \( \top \) value
      - In terms of gen/kill dfa, this specifies the initial facts assumed at each statement
  - Transfer functions
    - In terms of gen/kill dfa, this defines gen and kill
  - Facts at entry (for forward) or exit (for backward)
    - In terms of gen/kill dfa, this defines set of facts for entry or exit node
Four Analyses as Lattices \((P, \leq)\)

- **Available expressions**
  - Forward analysis
  - \(P = \text{sets of expressions}\)
  - \(S_1 \cap S_2 = S_1 \cap S_2\)
  - \(\text{Top} = \text{set of all expressions}\)
  - \(\text{Entry facts} = \emptyset = \text{no expressions available at entry}\)

- **Reaching Definitions**
  - Forward analysis
  - \(P = \text{set of definitions (assignment statements)}\)
  - \(S_1 \cap S_2 = S_1 \cup S_2\)
  - \(\text{Top} = \text{empty set}\)
  - \(\text{Entry facts} = \emptyset = \text{no definitions reach entry}\)
Four Analyses as Lattices \((P, \leq)\)

- **Very busy expressions**
  - Backward analysis
  - \(P = \text{sets of expressions}\)
  - \(S_1 \cap S_2 = S_1 \cap S_2\)
  - \(\text{Top} = \text{set of all expressions}\)
  - \(\text{Exit facts} = \emptyset = \text{no expressions busy at exit}\)

- **Live variables**
  - Backward analysis
  - \(P = \text{set of variables}\)
  - \(S_1 \cap S_2 = S_1 \cup S_2\)
  - \(\text{Top} = \emptyset = \text{empty set}\)
  - \(\text{Exit facts} = \emptyset = \text{no variables live at exit}\)
Constant Propagation

• Idea: maintain possible value of each variable

\[ L_a = \ldots \lor a=-1 \lor a=0 \lor a=1 \ldots \]

- \( \top \) = initial value = haven’t seen assignment to \( a \) yet
- \( \bot \) = multiple different possible values for \( a \)

• DFA definition:
  - Forward analysis
  - Lattice = \( L_a \times L_b \times \ldots \) (for all variables in program)
    - I.e., maintain one possible value of each variable
  - Initial facts (at entry) = \( \top \) (variables all unassigned)
Monotonicity and Desc. Chain

• A function $f$ on a partial order is monotonic (or order preserving) if
  
  $$x \leq y \Rightarrow f(x) \leq f(y)$$

  - Examples
    - $\lambda x. x+1$ on partial order $(\mathbb{Z}, \leq)$ is monotonic
    - $\lambda x. -x$ on partial order $(\mathbb{Z}, \leq)$ is not monotonic

• Transfer functions in gen/kill DFA are monotonic
  
  - $\text{temp} = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))$
    - Holds because $\text{gen}(s)$ and $\text{kill}(s)$ are fixed
    - Thus, if we shrink $\text{in}(s)$, $\text{temp}$ can only shrink

• A descending chain in a lattice is a sequence
  
  $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$

  - Height of lattice = length of longest descending chain
Monotonicity and Transfer Fns

• If $f_s$ is monotonic, how often can we apply this step?

$$\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s')$$
$$\text{temp} = f_s(\text{in}(s))$$

- **Claim:** $\text{out}(s)$ only shrinks
  - **Proof:** $\text{out}(s)$ starts out as $\top$
  - Assume $\text{out}(s')$ shrinks for all predecessors $s'$ of $s$
  - Then $\bigcap_{s' \in \text{pred}(s)} \text{out}(s')$ shrinks
  - Since $f_s$ monotonic, $f_s(\bigcap_{s' \in \text{pred}(s)} \text{out}(s'))$ shrinks
Termination

• Suppose we have a DFA with
  - Finite height lattice
  - Monotonic transfer functions

• Then, at every step in DFA we
  - Remove a statement from the worklist, and/or
  - Strictly decrease some dataflow fact at a program point
    - (By monotonicity)
    - Only add new statements to worklist after strict decrease
  - ⇒ termination! (by finite height)

• Moreover, must terminate in $O(nk)$ time
  - $n = \#$ of statements in program
  - $k = \text{height of lattice}$
  - (assumes meet operation takes $O(1)$ time)
Fixpoints

• We always start with $\top$
  - E.g., every expr is available, no defns reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis
    - = true of fewest number of states

• Revise as we encounter contradictions
  - Always move down in the lattice (with meet)

• Result: A greatest fixpoint solution of the data flow equations
Least vs. Greatest Fixpoints

• Dataflow tradition: Start with $\top$, use $\sqcap$
  - Computes a greatest fixpoint
  - Technically, rather than a lattice, we only need a finite height meet semilattice with top
    - finite height meet semilattice with top
    - $(meet\ semilattice = a \sqcap b$ defined on any $a,b$ but $a \sqcup b$ may not be defined)

• Denotational semantics trad.: Start with $\bot$, use $\sqcup$
  - Computes least fixpoint

• So, direction of DFA may depend on community author comes from...
Distributive Dataflow Problems

- A monotonic transfer function $f$ also satisfies

$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- Proof: By monotonicity, $f(x \sqcap y) \leq f(x)$ and $f(x \sqcap y) \leq f(y)$, i.e., $f(x \sqcap y)$ is a lower bound of $f(x)$ and $f(y)$. But then since $\sqcap$ is the greatest lower bound, $f(x \sqcap y) \leq f(x) \sqcap f(y)$.

- A transfer function $f$ is *distributive* if it satisfies

$$f(x \sqcap y) = f(x) \sqcap f(y)$$

- Notice this is stronger than monotonicity
Benefit of Distributivity

• Suppose we have the following CFG with four statements, a, b, c, d, with transfer fns \( f_a, f_b, f_c, f_d \)

\[
\begin{align*}
\text{out}(a) &= f_a(\text{in}(a)) \\
\text{out}(b) &= f_b(\text{in}(b)) \\
\text{out}(c) &= f_c(\text{out}(a) \sqcap \text{out}(b)) = f_c(f_a(\text{in}(a)) \sqcap f_b(\text{in}(b))) = f_c(f_c(f_a(\text{in}(a)))) \sqcap f_c(f_c(f_b(\text{in}(b)))) \\
\text{out}(d) &= f_d(\text{out}(c)) = f_d(f_c(f_a(\text{in}(a)))) \sqcap f_d(f_c(f_b(\text{in}(b)))) = f_d(f_c(f_c(f_a(\text{in}(a)))) \sqcap f_d(f_c(f_c(f_b(\text{in}(b))))))
\end{align*}
\]

Then joins lose no information!
Ideally, we would like to compute the *meet over all paths* (MOP) solution:

- If $p$ is a path through the CFG, let $f_p$ be the composition of transfer functions for the statements along $p$
- Let $\text{path}(s)$ be the set of paths from the entry to $s$
- Define

$$MOP(s) = \bigcap_{p \in \text{path}(s)} f_p(\top)$$

- I.e., $MOP(s)$ is the set of dataflow facts if we separately apply the transfer functions along every path (assuming $\top$ is the initial value at the entry) and then apply $\cap$ to the result
- This is the best we could possibly do if we want one data flow fact per program point and we ignore conditional tests along the path

If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution!
What Problems are Distributive?

• Analyses of how the program computes
  ▪ Live variables
  ▪ Available expressions
  ▪ Reaching definitions
  ▪ Very busy expressions

• All gen/kill problems are distributive
A Non-Distributive Example

• Constant propagation

\[
\begin{align*}
x &:= 1 \\
y &:= 2 \\
z &:= x + y
\end{align*}
\]

\[
\begin{align*}
x &:= 2 \\
y &:= 1
\end{align*}
\]

- \(\{x=1, y=2\} \cap \{x=2, y=1\} = \{x=\perp, y=\perp\}\)
- The join at \(\text{in}(z:=x+y)\) loses information
- But in fact, \(z=3\) every time we run this program

• In general, analysis of \textit{what} the program computes is not distributive
Basic Blocks

• Recall a *basic block* is a sequence of statements s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

• In some data flow implementations,
  - Compute gen/kill for each basic block as a whole
    - Compose transfer functions
  - Store only in/out for each basic block
  - Typical basic block ~5 statements
    - At least, this used to be the case...
Order Matters

• Assume forward data flow problem
  ▪ Let $G = (V, E)$ be the CFG
  ▪ Let $k$ be the height of the lattice

• If $G$ acyclic, visit in topological order
  ▪ Visit head before tail of edge

• Running time $O(|E|)$
  ▪ No matter what size the lattice
Order Matters — Cycles

• If $G$ has cycles, visit in reverse postorder
  ▪ Order from depth-first search
  ▪ (Reverse for backward analysis)

• Let $Q = \max$ # back edges on cycle-free path
  ▪ Nesting depth
  ▪ Back edge is from node to ancestor in DFS tree

• If $\forall x.f(x) \leq x$ (sufficient, but not necessary), then running time is $O((Q+1)|E|)$
  ▪ Proportional to structure of CFG rather than lattice
Flow-Sensitivity

• Data flow analysis is flow-sensitive
  ▪ The order of statements is taken into account
  ▪ I.e., we keep track of facts per program point

• Alternative: Flow-insensitive analysis
  ▪ Analysis the same regardless of statement order
  ▪ Standard example: types
    - /* x : int */ x := ... /* x : int */
Data Flow Analysis and Functions

• What happens at a function call?
  ■ Lots of proposed solutions in data flow analysis literature

• In practice, only analyze one procedure at a time

• Consequences
  ■ Call to function kills all data flow facts
  ■ May be able to improve depending on language, e.g., function call may not affect locals
**More Terminology**

- An analysis that models only a single function at a time is *intraprocedural*.
- An analysis that takes multiple functions into account is *interprocedural*.
- An analysis that takes the whole program into account is *whole program*.

- Note: *global* analysis means “more than one basic block,” but still within a function.
  - Old terminology from when computers were slow...
Data Flow Analysis and The Heap

• Data Flow is good at analyzing local variables
  ▶ But what about values stored in the heap?
  ▶ Not modeled in traditional data flow

• In practice: $x := e$
  ▶ Assume all data flow facts killed (!)
  ▶ Or, assume write through $x$ may affect any variable whose address has been taken

• In general, hard to analyze pointers
Proebsting’s Law

• Moore’s Law: Hardware advances double computing power every 18 months.

• Proebsting’s Law: Compiler advances double computing power every 18 years.
  - Not so much bang for the buck!
DFA and Defect Detection

- LCLint - Evans et al. (UVa)
- METAL - Engler et al. (Stanford, now Coverity)
- ESP - Das et al. (MSR)
- FindBugs - Hovemeyer, Pugh (Maryland)
  - For Java. The first three are for C.

- Many other one-shot projects
  - Memory leak detection
  - Security vulnerability checking (tainting, info. leaks)