Dataflow analysis (ctd.)
Available expressions

- Determine which expressions have already been evaluated at each point.

- A expression $x+y$ is *available at point* $p$ if every path from the entry to $p$ evaluates $x+y$ and after the last such evaluation prior to reaching $p$, there are no assignments to $x$ or $y$.

- Used in:
  - global common subexpression elimination
Example

\[
\begin{align*}
  x &= x+y \\
  z &= a+b
\end{align*}
\]

Generates \( a+b \), Kills \( x+y \), \( w^x \), etc. Kills \( z-w \), \( x+z \), etc.
Available expressions

- What is safe?
  - To assume that an expression is not available at some point even if it may be.
  - The computed set of available expressions at point p will be a subset of the actual set of available expressions at p.
  - The computed set of unavailable expressions at point p will be a superset of the actual set of unavailable expressions at p.
  - Goal: make the set of available expressions as large as possible (i.e. as close to the actual set as possible).
Available expressions

• How are the **gen** and **kill** sets defined?
  – \( \text{gen}[B] = \{\text{expressions evaluated in } B \text{ without subsequently redefining its operands}\} \)
  – \( \text{kill}[B] = \{\text{expressions whose operands are redefined in } B \text{ without reevaluating the expression afterwards}\} \)

• What is the direction of the analysis?
  – forward
  – \( \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \)
Available expressions

• What is the confluence operator?
  – intersection
  – \( \text{in}[B] = \cap \text{out}[P] \), over the predecessors \( P \) of \( B \)

• How do we initialize?
  – Start with emptyset!
Available Expressions: equations

\[ AE_{\text{entry}}(p) = \begin{cases} \emptyset & \text{for initial point } p \\ \bigcap \{ AE_{\text{exit}}(q) \mid (q,p) \text{ in the CFD} \} \end{cases} \]

\[ AE_{\text{exit}}(p) = \text{gen}_{AE}(p) \cup (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \]
Equations

<table>
<thead>
<tr>
<th>n</th>
<th>$\text{kill}_{AE}(n)$</th>
<th>$\text{gen}_{AE}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>${a*b}$</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>4</td>
<td>${a+b, a*b, a+1}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
</tr>
</tbody>
</table>

$AE_{entry}(p) = \emptyset$ for initial point $p$

$AE_{entry}(p) = \bigcap \{ AE_{exit}(q) | (q, p) \text{ in CFD} \}$

$AE_{exit}(p) = (AE_{entry}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$

$AE_{entry}(1) = \emptyset$

$AE_{entry}(2) = AE_{exit}(1)$

$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$

$AE_{entry}(4) = AE_{exit}(3)$

$AE_{entry}(5) = AE_{exit}(4)$

$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$

$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$

$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$

$AE_{exit}(4) = AE_{entry}(4) - \{a+b, a*b, a+1\}$

$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$
Solution

$AE_{entry}(1) = \emptyset$
$AE_{entry}(2) = AE_{exit}(1)$
$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$
$AE_{entry}(4) = AE_{exit}(3)$
$AE_{entry}(5) = AE_{exit}(4)$

$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$
$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$
$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$
$AE_{exit}(4) = AE_{entry}(4) - \{a+b, a*b, a+1\}$
$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$

<table>
<thead>
<tr>
<th>n</th>
<th>$AE_{entry}(n)$</th>
<th>$AE_{exit}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>2</td>
<td>${a+b}$</td>
<td>${a+b, a*b}$</td>
</tr>
<tr>
<td>3</td>
<td>${a+b}$</td>
<td>${a+b}$</td>
</tr>
<tr>
<td>4</td>
<td>${a+b}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
</tr>
</tbody>
</table>
Result

• \([x:=a+b]^1; [y:=a*b]^2; \text{while} [y>a+b]^3 \text{do} \{ [a:=a+1]^4; [x:=a+b]^5 \}\]

<table>
<thead>
<tr>
<th>n</th>
<th>(\text{AE}_{\text{entry}}(n))</th>
<th>(\text{AE}_{\text{exit}}(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>{a+b}</td>
</tr>
<tr>
<td>2</td>
<td>{a+b}</td>
<td>{a+b, a*b}</td>
</tr>
<tr>
<td>3</td>
<td>{a+b}</td>
<td>{a+b}</td>
</tr>
<tr>
<td>4</td>
<td>{a+b}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>5</td>
<td>(\emptyset)</td>
<td>{a+b}</td>
</tr>
</tbody>
</table>

• Even though the expression a is redefined in the cycle (in 4), the expression \(a+b\) is always available at the entry of the cycle (in 3).

• Viceversa, \(a*b\) is available at the first entry of the cycle but it is killed before the next iteration (in 4).
Very Busy Expressions

- Determine whether an expression is evaluated in all paths from a point to the exit.
- An expression $e$ is very busy at point $p$ if no matter what path is taken from $p$, $e$ will be evaluated before any of its operands are defined.

- Used in:
  - Code hoisting
    - If $e$ is very busy at point $p$, we can move its evaluation at $p$. 
Example

if \([a>b]\) then \((x:=b-a) ; \; (y:=a-b)\) else \((y:=b-a) ; \; (x:=a-b)\)

The two expressions \(a-b\) and \(b-a\) are both very busy in program point 1.
What is safe?
- To assume that an expression is not very busy at some point even if it may be.
- The computed set of very busy expressions at point p will be a subset of the actual set of available expressions at p.
- Goal: make the set of very busy expressions as large as possible (i.e. as close to the actual set as possible).
Very Busy Expressions

- How are the **gen** and **kill** sets defined?
  - \( \text{gen}[B] = \{ \text{all expressions evaluated in } B \text{ before any definitions of their operands} \} \)
  - \( \text{kill}[B] = \{ \text{all expressions whose operands are defined in } B \text{ before any possible re-evaluation} \} \)

- What is the direction of the analysis?
  - backward
  - \( \text{in}[B] = \text{gen}[B] \cup (\text{out}[B] - \text{kill}[B]) \)
• What is the confluence operator?
  – intersection
  – $\text{out}[B] = \cap \text{in}[S], \text{ over the successors } S \text{ of } B$
Very Busy Expressions: equations

\[
\begin{align*}
\text{VB}_{\text{exit}}(p) &= \begin{cases}
\emptyset & \text{if } p \text{ is final} \\
\cap \{ \text{VB}_{\text{entry}}(q) \mid (p,q) \text{ in the CFD} \}
\end{cases} \\
\text{VB}_{\text{entry}}(p) &= (\text{VB}_{\text{exit}}(p) \setminus \text{kill}_{\text{VB}}(p)) \cup \text{gen}_{\text{VB}}(p)
\end{align*}
\]
<table>
<thead>
<tr>
<th>n</th>
<th>kill_{VB}(n)</th>
<th>gen_{VB}(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>{b-a}</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>{a-b}</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>{b-a}</td>
</tr>
<tr>
<td>5</td>
<td>∅</td>
<td>{a-b}</td>
</tr>
</tbody>
</table>

VB_{entry}(1)=VB_{exit}(1)
VB_{entry}(2)=VB_{exit}(2) \cup \{b-a\}
VB_{entry}(3)=\{a-b\}
VB_{entry}(4)=VB_{exit}(4) \cup \{b-a\}
VB_{entry}(5)=\{a-b\}

VB_{exit}(1)= VB_{entry}(2) \cap VB_{entry}(4)
VB_{exit}(2)= VB_{entry}(3)
VB_{exit}(3)= \emptyset
VB_{exit}(4)= VB_{entry}(5)
VB_{exit}(5)= \emptyset

\begin{align*}
\text{v} & = a > b \\
\text{x} & = b - a \\
\text{y} & = b - a \\
\text{y} & = a - b \\
\text{x} & = a - b
\end{align*}
if \([a > b]\) then \([x := b - a] ; [y := a - b]\) else \([y := b - a] ; [x := a - b]\)

<table>
<thead>
<tr>
<th>n</th>
<th>(\text{VB}_{\text{entry}}(n))</th>
<th>(\text{VB}_{\text{exit}}(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a-b, b-a}</td>
<td>{a-b, b-a}</td>
</tr>
<tr>
<td>2</td>
<td>{a-b, b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>3</td>
<td>{a-b}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>4</td>
<td>{a-b, b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>5</td>
<td>{a-b}</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
Dataflow analysis: a general framework
Dataflow Analysis

• Compile-time reasoning about run-time values of variables or expressions

• At different program points
  – Which assignment statements produced value of variable at this point?
  – Which variables contain values that are no longer used after this program point?
  – What is the range of possible values of variable at this program point?
Program Representation

- **Control Flow Graph**
  - Nodes $N$ – statements of program
  - Edges $E$ – flow of control
    - $\text{pred}(n) =$ set of all predecessors of $n$
    - $\text{succ}(n) =$ set of all successors of $n$
  - Start node $n_0$
  - Set of final nodes $N_{\text{final}}$
Program Points

• One program point before each node
• One program point after each node
• Join point – point with multiple predecessors
• Split point – point with multiple successors
Basic Idea

- Information about program represented using values from algebraic structure
- Analysis produces a value for each program point
- Two flavors of analysis
  - Forward dataflow analysis
  - Backward dataflow analysis
Forward Dataflow Analysis

• Analysis propagates values forward through control flow graph with flow of control
  – Each node $n$ has a transfer function $f_n$
    • Input – value at program point before node
    • Output – new value at program point after node
  – Values flow from program points after predecessor nodes to program points before successor nodes
  – At join points, values are combined using a merge function

• Canonical Example: Reaching Definitions
Backward Dataflow Analysis

- Analysis propagates values backward through control flow graph against flow of control
  - Each node $n$ has a transfer function $f_n$
    - Input – value at program point after node
    - Output – new value at program point before node
  - Values flow from program points before successor nodes to program points after predecessor nodes
  - At split points, values are combined using a merge function
- Canonical Example: Live Variables
Representing the property of interest

• 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

• Will use \( \lor \) to combine values at control-flow join points
Transfer Functions

- Transfer function $f_n : P \rightarrow P$ for each node $n$ in control flow graph
- $f_n$ models effect of the node on the program information
Transfer Functions

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

- Identity function $i \in F$
- $F$ must be closed under composition: $
\forall f, g \in F. \text{ the function } h(x) = f(g(x)) \in F$
- Each $f \in F$ must be monotone: $x \leq y$ implies $f(x) \leq f(y)$

- Sometimes all $f \in F$ are distributive: $f(x \lor y) = f(x) \lor f(y)$
- Distributivity implies monotonicity
Distributivity Implies Monotonicity

• Proof of distributivity implies monotonicity

• Assume \( f(x ∨ y) = f(x) ∨ f(y) \)

• Must show:
  \( x ≤ y \) implies \( f(x) ≤ f(y) \), and this is equivalent to show that \( x ∨ y = y \) implies \( f(x) ∨ f(y) = f(y) \)

  \[
  f(y) = f(x ∨ y) \quad \text{(by applying \( f \) to both sides)}
  \]

  \[
  = f(x) ∨ f(y) \quad \text{(by distributivity)}
  \]
Forward Dataflow Analysis

- Simulates execution of program forward with flow of control
- For each node $n$, have
  - $\text{in}_n$ – value at program point before $n$
  - $\text{out}_n$ – value at program point after $n$
  - $f_n$ – transfer function for $n$ (given $\text{in}_n$, computes $\text{out}_n$)
- Require that solution satisfy
  - $\forall n. \text{out}_n = f_n(\text{in}_n)$
  - $\forall n \neq n_0. \text{in}_n = \bigvee \{ \text{out}_m . m \in \text{pred}(n) \}$
  - $\text{in}_{n_0} = I$,
    where I summarizes information at start of program
Worklist Algorithm for Solving Forward Dataflow Equations

for each $n$ do $\text{out}_n := f_n(\bot)$

$\text{in}_{n_0} := I; \text{out}_{n_0} := f_{n_0}(I)$

worklist := $N - \{ n_0 \}$

while worklist $\neq \emptyset$ do

remove a node $n$ from worklist

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

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

if $\text{out}_n$ changed then

worklist := worklist $\cup$ succ($n$)
Correctness Argument

• Why result satisfies dataflow equations?
• Whenever process a node n, the algorithm ensures that $out_n = f_n(in_n)$
• Whenever $out_m$ changes, the algorithm puts $succ(m)$ on worklist.
  Consider any node $n \in succ(m)$. It will eventually come off worklist and the algorithm will set
  \[
  in_n := \bigvee \{ out_m \cdot m \in pred(n) \}
  \]
  to ensure that $in_n = \bigvee \{ out_m \cdot m \in pred(n) \}$
• 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 the lattice enjoys the ascending chain property, the algorithm terminates
  – Algorithm terminates for finite lattices
  – For lattices without ascending chain property, we may 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

- \( P = \) powerset of set of all definitions in program (all subsets of set of definitions in program)
- \( \lor = \bigcup \) (order is \( \subseteq \))
- \( \bot = \emptyset \)
- \( I = \text{in}_{n_0} = \bot \)
- \( F = \) all functions \( f \) 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
  - \( f(x) = \text{GEN} \cup (x-\text{KILL}) \)
Does Reaching Definitions Satisfy the Framework Constraints?

• $\subseteq$ 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)

  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 $a \cup (x - b)$

$$f_1(f_2(x)) = a_1 \cup ((a_2 \cup (x-b_2)) - b_1)$$
$$= a_1 \cup ((a_2 - b_1) \cup ((x-b_2) - b_1))$$
$$= (a_1 \cup (a_2 - b_1)) \cup ((x-b_2) - b_1))$$
$$= (a_1 \cup (a_2 - b_1)) \cup (x-(b_2 \cup b_1))$$

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

- $P = \text{powerset of set of all expressions in program (all subsets of set of expressions)}$
- $\mathbin{\lor} = \cap$ (order is $\supseteq$)
- $\bot = P$
- $I = \text{in}_{n_0} = \emptyset$
- $F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b)$
  - $b$ 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. Structure of analysis varies according to way analysis used.
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
  - $\forall n. \ in_n = f_n(out_n)$
  - $\forall n \not\in N_{\text{final}}. \ out_n = \lor \{ \ in_m. \ m \ in \ succ(n) \}$
  - $\forall n \in N_{\text{final}} = out_n = O$

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 $\text{out}_n := O; \text{in}_n := f_n(O)$
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
        worklist := worklist $\cup$ pred($n$)
Live Variables

- $P = \text{powerset of set of all variables in program (all subsets of set of variables in program)}$
- $\lor = \cup \text{ (order is } \subseteq\text{)}$
- $\bot = \emptyset$
- $O = \emptyset$
- $F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b)$
  - $b$ is set of variables that node kills
  - $a$ is set of variables that node reads