Global Data_Flow Analysis

Examples:

Reaching Definitions:
Which DEFINITIONs reach which USEs?

LIVE Variable Analysis:
Which variables are live at a given point, P?

Global Sub-Expression Elimination:
Which expressions reach point P
and do not need to be re-computed?

Copy Propagation:
Which copies reach point P?
Can we do copy propagation?

Terminology

A "point"
between two adjacent statements in a basic block,
or directly before the basic block,
or directly after the basic block.

© Harry H. Porter, 2006
Terminology

A "point" between two adjacent statements in a basic block, or directly before the basic block, or directly after the basic block.

A "path" is a sequence of points from $P_1$ to $P_N$ such that control could flow from $P_1$ to $P_N$. The path may traverse several blocks.

Reaching Definitions

A "definition" of variable $x$ is a statement that assigns to $x$ (or might assign to $x$).

Ambiguous Definitions -- Might assign
Unambiguous Definitions -- Will definitely assign

Examples

- $x := \ldots;$  
  Unambiguous; will definitely change $x$
- `read (x);`
- `call foo (... x ...)`
- `call foo ();`
- `*p := \ldots;;`
- `y := \ldots;`
“Killing” Definitions

A definition is “killed” along a path...
if there is an unambiguous definition of the variable.

\[ \begin{align*}
  & x := a + b \\
  & c := b \cdot d \\
  & e := a - x \\
  & b := a + e \\
  & c := x + a \\
\end{align*} \]

This definition... is killed by this statement... before it reaches this point

A definition D “reaches” a point P...
if there is a path from D to P along which D is not killed.

If “x” is defined at D, then the value given to “x” might be the value of “x” at point P.

When D reaches P, it means...
The value of “x” might reach P at runtime.
A definition D “reaches” a point P... if there is a path from D to P along which D is not killed.

If “x” is defined at D, then the value given to “x” might be the value of “x” at point P.

When D reaches P, it means...
The value of “x” might reach P at runtime.

Does \( D_1 \) reach P? YES
Does \( D_2 \) reach P? NO
Safe, Conservative Estimates

Will the value of \( x \) reach point \( P \)?

The runtime value of variables may cause some paths to... NEVER be taken.

It may be the case that...

\( \text{In ALL executions, control ALWAYS passes through B2...} \)

\( \text{D may get killed in every execution!} \)

\( \text{The value of “x” may never reach point \( P \)!} \)

Nevertheless, we say “\( D \) reaches \( P \)”.

\[ \begin{align*}
\text{D}_1: & \quad x := a+b \\
\text{D}_2: & \quad y := c+d \\
\text{...} & \\
\text{B}_2: & \quad x := c, y := e-d \\
\text{B}_3: & \quad y := b+1 \\
\text{B}_4: & \quad d := a \cdot b, b := x + y
\end{align*} \]

It is undecidable (in general) to determine statically which paths will be taken at runtime.

USE-DEFINITION Chains (U-D Chains)

For each USE of some variable “\( x \)”...
build a list of all the DEFINITIONs of “\( x \)” that reach this USE.

\[ \begin{align*}
\text{...} & \quad x := a+b \\
\text{...} & \quad x := c+d \\
\text{...} & \quad d := (x) \cdot b \\
\text{...} & \quad c := e - x \\
\text{...} & \quad x := b + c \\
\text{...} & \quad g := a - 1 \\
\text{...} & \quad b := a + x
\end{align*} \]
USE-DEFINITION Chains (U-D Chains)

For each USE of some variable “x”...
build a list of all the DEFINITIONs of “x”
that reach this USE.

\[
\begin{align*}
\text{...} & \quad x := a + b \\
\text{...} & \quad x := c + d \\
\text{...} & \quad d := x \times b \\
\text{...} & \quad c := e - x \\
\text{...} & \quad x := b + c \\
\text{...} & \quad g := a - 1 \\
\text{...} & \quad b := a + x \\
\end{align*}
\]
If we can deduce that the set of definitions reaching this point contains \textit{ONLY} the assignment $D$ to “$x$”, then it is okay to substitute 5 for “$x$” here.

\begin{itemize}
\item $D: x := 5$
\item $y := x \times 2$
\end{itemize}

\textbf{DEFINITION-USE Chains (D-U Chains)}

A variable is \textit{used} at statement $S$ if its value \textit{may be} required.

For each \textit{definition} of a variable...
compute a list of all possible \textit{uses} of that variable.

\begin{itemize}
\item $x := a + b$
\item $y := x \times 3$
\item $z := a + x$
\item $z := x + y$
\item $x := x + 1$
\end{itemize}

... but not here
If we can deduce that the definition $D$ has *NO POSSIBLE USES* then $D$ is “DEAD” (useless code) and can be eliminated!

\[
D: x := 5
\]

**The Universe**

$\mathcal{U} = \text{Universe} = \text{the set of all DEFINITIONs in the program / CFG}$

Number them $D_1, D_2, D_3, \ldots$

*Example:*

\[
B_1 \quad D_i: i := m-1
D_j: j := n
D_a: a := w
\]

\[
B_2 \quad D_i: i := i+1
D_j: j := j-1
\text{if...}
\]

\[
B_3 \quad D_a: a := y
\]

\[
B_4 \quad D_i: i := z
\text{if...}
\]
Representing Sets

We will work with sets.
How to represent?
Each set is represented with a Bit Vector

<table>
<thead>
<tr>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
<th>$D_5$</th>
<th>$D_6$</th>
<th>$D_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Example

$A = \{D_2, D_4, D_7\}$

$A' = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$

How to compute set operations?

- **Set Union**
  
  $A \cup B \Rightarrow$

- **Set Intersection**
  
  $A \cap B \Rightarrow$

- **Set Difference**
  
  $A - B \Rightarrow$
Representing Sets

We will work with sets.
How to represent?
Each set is represented with a Bit Vector

<table>
<thead>
<tr>
<th>D_1</th>
<th>D_2</th>
<th>D_3</th>
<th>D_4</th>
<th>D_5</th>
<th>D_6</th>
<th>D_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Example

A = \{ D_2, D_4, D_7 \}
A' =

How to compute set operations?

Set Union

A \cup B \Rightarrow A' \text{ or } B'

Set Intersection

A \cap B \Rightarrow A' \text{ and } B'

Set Difference

A - B \Rightarrow A' \text{ and } (\text{not } B')

Approach

Figure out what happens in each basic block...

\text{GEN}[B] =
- The set of definitions appearing in block B
  which reach the end of B
  (without being KILLED before the end of the block)

\text{KILL}[B] =
- The set of definitions KILLED by statements in block B.
- If B contains an unambiguous definition of variable “x”,
  then add all definitions of “x” to \text{KILL}[B].
  (unless the definition D of “x” also occurs in B and
  there are no unambiguous definitions between D
  and the end of B).

Use this info to do the entire flow graph...

Using DATA FLOW EQUATIONS
**Example of GEN [B]**

Consider this Basic Block:

\[
\begin{array}{|l|}
\hline
D_5: & c := e - x \\
D_6: & x := b + c \\
D_7: & x := a - 4 \\
\hline
\end{array}
\]

Consider \(D_5\), a definition of “c”...
Add \(D_5\) to GEN [B].

Consider \(D_6\), a definition of “x”...
But this is KILLED before the end of the block.

Consider \(D_7\), a definition of “x”...
Add \(D_7\) to GEN [B].

\[
\text{GEN [B]} = \{D_5, D_7\}
\]

---

**Example of KILL [B]**

Consider this Basic Block:

\[
\begin{array}{|l|}
\hline
D_5: & c := e - x \\
D_6: & x := b + c \\
D_7: & x := a - 4 \\
\hline
\end{array}
\]

Consider \(D_5\), an unambiguous definition of “c”...
Add all other definitions of “c” to KILL [B].

(Except, do not add \(D_5\) itself, since this definition “makes it to the end of the block”.)

Consider \(D_7\), an unambiguous definition of “x”...
Add all other definitions of “x” to KILL [B] 
(Except, do not add \(D_7\) itself, since this definition “makes it to the end of the block”.)
Overview of the Computation

For every point in the program... we want to know which definitions can reach that point.

We will compute the set of definitions that can reach the beginning of a basic block:

\[ \text{IN} \{ B \} \]

Then, using GEN \{ B \} and KILL \{ B \}, we will compute the set of definitions reaching the end of the basic block:

\[ \text{OUT} \{ B \} \]

Then we will use OUT \{ B \} to compute the set of definitions that can reach other basic blocks.

... And we will repeat, until we learn which definitions could possibly reach which blocks.

In the text: \( \text{Reaches()} \)

The Data Flow Algorithm

Approach:
Build the \( \text{IN} \) and \( \text{OUT} \) sets simultaneously, by successive approximations!

Given:
A control flow graph of basic blocks.

Assume:
\( \text{GEN}[B] \) and \( \text{KILL}[B] \) have already be computed for each basic block.

Output:
\( \text{IN}[B] \) and \( \text{OUT}[B] \) for each basic block.
Start by setting \( \text{IN}[B] \) to \( \{\} \) for each basic block.
Then compute \( \text{OUT}[B] \) from the previous estimate of \( \text{IN}[B] \).
Finally, propagate \( \text{OUT}[B] \) to the \( \text{IN}[B'] \) for all successor blocks to \( B \).
Repeat, until no more changes.
As the definitions “flow through the graph”, the \( \text{IN} \) and \( \text{OUT} \) sets grow and grow.
The approximation gets closer and closer.

**Conservative:** May overestimate how far definitions will reach.
(i.e., the results may be larger than “truly” necessary.)

A Recurrence
(a set of simultaneous equations)

\[
\text{IN}[B] := \bigcup_{P \text{ is a predecessor of } B} \text{OUT}[P]
\]
\[
\text{OUT}[B] := \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B])
\]

\[
\sum_{0 < i < N} f(i)
\]
for each block $B$ do
  \[ \text{OUT}[B] := \text{GEN}[B] \]
endfor

\[ \text{OUT}[B] := \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]) \]

while change do
  \[ \text{OUT}[B] := \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]) \]
endwhile

\[ \text{OUT}[B] := \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]) \]
**Example**

\[
\begin{align*}
&D_1: i := m - 1 \\
&D_2: j := n \\
&D_3: a := w \\
&D_4: i := i + 1 \\
&D_5: j := j - 1 \\
&D_6: a := y \\
&D_7: i := z
\end{align*}
\]

**GEN[B_1]** = \{D_1, D_2, D_3\}

**KILL[B_1]** = \{D_4, D_5, D_6, D_7\}

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0
\end{array}
\]

**GEN[B_2]** = \{D_4, D_5\}

**KILL[B_2]** = \{D_6, D_7\}

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0
\end{array}
\]

**GEN[B_3]** = \{D_6\}

**KILL[B_3]** = \{D_7\}

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{array}
\]

**GEN[B_4]** = \{D_7\}

**KILL[B_4]** = \{D_1, D_4\}

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{array}
\]

**OUT**

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{array}
\]

**IN**

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{array}
\]
Example

\[ \text{D}_1: i := m-1 \]
\[ \text{D}_2: j := n \]
\[ \text{D}_3: a := w \]
\[ \text{D}_4: i := i+1 \]
\[ \text{D}_5: j := j-1 \]
\[ \text{if}... \]
\[ \text{D}_6: i := z \]
\[ \text{if}... \]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{B}_1 & \text{B}_2 & \text{B}_3 & \text{B}_4 \\
\hline
\text{OUT} & 111 & 0000 & 000 & 1100 & 000 & 0010 & 000 & 0001 \\
\text{IN} & 000 & 0000 & 111 & 0001 & 000 & 1100 & 000 & 1110 \\
\text{OUT} & 111 & 0000 & 001 & 1100 & 000 & 1110 & 000 & 0111 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{B}_1 & \text{B}_2 & \text{B}_3 & \text{B}_4 \\
\hline
\text{OUT} & 111 & 0000 & 000 & 1100 & 000 & 0010 & 000 & 0001 \\
\text{IN} & 000 & 0000 & 111 & 0001 & 000 & 1100 & 000 & 1110 \\
\text{OUT} & 111 & 0000 & 001 & 1100 & 000 & 1110 & 000 & 0111 \\
\hline
\end{array}
\]
Example

\[
\begin{align*}
D_1: & \quad i := m - 1 \\
D_2: & \quad j := n \\
D_3: & \quad a := w \\
D_4: & \quad i := i + 1 \\
D_5: & \quad j := j - 1 \\
D_6: & \quad a := y \\
D_7: & \quad i := z
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c}
B_1 & B_2 & B_3 & B_4 \\
\hline
\text{OUT} & 111 & 0000 & 000 & 1100 \\
\text{IN} & 000 & 0000 & 111 & 0001 \\
\text{OUT} & 111 & 0000 & 001 & 1110 \\
\text{IN} & 000 & 0000 & 111 & 0111 \\
\text{OUT} & 111 & 0000 & 001 & 1110 \\
\text{IN} & 000 & 0000 & 111 & 0111 \\
\end{array}
\]

© Harry H. Porter, 2006
Example

\[ \begin{align*}
D_1 &: \ i := m - 1 \\
D_2 &: \ j := n \\
D_3 &: \ a := w \\
D_4 &: \ i := i + 1 \\
D_5 &: \ j := j - 1 \\
D_6 &: \ a := y \\
D_7 &: \ i := z
\end{align*} \]

\[ \begin{array}{c|c|c|c|c}
\text{B}_1 & \text{B}_2 & \text{B}_3 & \text{B}_4 \\
\hline
\text{IN} & 000 & 000 & 001 & 001 \\
\text{OUT} & 111 & 000 & 001 & 001 \\
\hline
\text{IN} & 000 & 001 & 001 & 001 \\
\text{OUT} & 111 & 000 & 001 & 001 \\
\hline
\text{IN} & 000 & 000 & 001 & 001 \\
\text{OUT} & 111 & 000 & 001 & 001 \\
\hline
\text{IN} & 000 & 000 & 001 & 001 \\
\text{OUT} & 111 & 000 & 001 & 001 \\
\end{array} \]
This algorithm converges.
OUT[B] never decreases...
Once in OUT[B] a definition stays there.
Eventually, no changes will be made to OUT[B].

An upper bound on the “while” loop?
Number of nodes in the flow graph.
Each iteration propagates reaching definitions.

The “while” loop will converge quickly
...if you select a good order for the nodes in the “for” loop.

This algorithm is efficient in practice.

LIVE Variable Analysis

A similar Data Flow Algorithm
Goal: Compute IN[] and OUT[]
However, it will work backwards!
(i.e., data will flow “upwards”, against the arrow directions)

Given:
Compute:
Which variables are LIVE here
Which variables are LIVE here
**LIVE Variable Analysis**

*Then:*
Compute the OUT set from all the IN sets of the block’s successors!

![Diagram showing variable analysis process](image)

© Harry H. Porter, 2006
**LIVE Variable Analysis**

*Then:*
Compute the OUT set from
all the IN sets of the block’s successors!

Info flows upwards!
“against” the flow graph edges
Definitions
Variable “x” is LIVE at some point P if its value might be used at some point later, on a path starting at P.

DEF [B] = the set of variables definitely assigned values in block B (prior to any use in B)

USE [B] = the set of variables whose values may be used in B (prior to any definitions of the variable)

IN [B] = the set of variables LIVE at the beginning of B
OUT [B] = the set of variables LIVE at the end of B

Note these re-definitions

Definitions
Variable “x” is LIVE at some point P if its value might be used at some point later, on a path starting at P.

DEF [B] = the set of variables definitely assigned values in block B (prior to any use in B)

USE [B] = the set of variables whose values may be used in B (prior to any definitions of the variable)

IN [B] = the set of variables LIVE at the beginning of B
OUT [B] = the set of variables LIVE at the end of B

Note these re-definitions
Recurrence Equations to be Solved:

\[
\begin{align*}
\text{IN}[B] & := \text{USE}[B] \cup (\text{OUT}[B] - \text{DEF}[B]) \\
\text{OUT}[B] & := \bigcup S \in \text{IN}[S] \quad S \text{ is a successor of } B
\end{align*}
\]

© Harry H. Porter, 2006
Algorithm to Compute LIVE Variables

Input:
Flow graph of basic blocks
DEF and USE for each block

Output:
OUT[B] = Live variables at end of B

Algorithm:
for each block B do
    IN[B] := {}
endfor
while changes occur for any IN set do
    for each block B do
        OUT[B] := \bigcup \{IN[S] \mid S \text{ is a successor of } B\}
        IN[B] := USE[B] \cup (OUT[B] - DEF[B])
    endfor
endwhile

Example
Compute LIVE variables
At the end of each block.

B1
a := 1
b := 2
a, b, e
B2
c := a+b
d := c-a
a, b, c, d, e
B3
d := b*d
a, b, c, d, e
B4
d := a+b
e := e+1
a, b, c, d, e
B5
b := a+b
e := c-a
a, b, d, e
B6
a := b*d
b := a-d
<none>
Computing Available Expressions

An “expression”:  
\[ x \oplus y \]

Binary expressions only  
Any operator: +, −, *, ...

Examples: a−b, w+x, y*4, ...

An expression is “available” at point P if every path to P computes it and there are no subsequent assignments to \( x \) or \( y \) (between the last evaluation of \( x \oplus y \) and P).

A block “generates” \( x \oplus y \) if it evaluates \( x \oplus y \) and does not subsequently assign to \( x \) or \( y \).

A block “kills” \( x \oplus y \) if it assigns to \( x \) or \( y \) without subsequently recomputing \( x \oplus y \).

Example

Which expressions are available?

\[
\begin{align*}
x & := y + z \\
y & := x - w \\
a & := w + z \\
z & := x - w \\
y & := y + z
\end{align*}
\]
Example

Which expressions are available?

\[ U = \{ y+z, x-w, w+z \} \]

\[ x := y + z \]
\[ y := x - w \]
\[ a := w + z \]
\[ z := x - w \]
\[ y := y + z \]

Avail = {}
Example

Which expressions are available?

\[ U = \{ y + z, x - w, w + z \} \]

\[
\begin{align*}
  x & := y + z \quad \text{Avail} = \{} \quad \text{Avail} = \{ y + z \} \\
  y & := x - w \quad \text{Avail} = \{} \\
  a & := w + z \\
  z & := x - w \\
  y & := y + z
\end{align*}
\]
Example

Which expressions are available?

\[ U = \{ y+z , x-w , w+z \} \]

\[ x := y + z \quad \text{Avail} = \{ y+z \} \]
\[ y := x - w \quad \text{Avail} = \{ x-w \} \]
\[ a := w + z \quad \text{Avail} = \{ x-w, w+z \} \]
\[ z := x - w \]
\[ y := y + z \quad \text{Avail} = \{ x-w \} \]

© Harry H. Porter, 2006
Example

Which expressions are available?

\[ U = \{ y+z , x-w , w+z \} \]

\[ x := y + z \quad \text{Avail} = \{ y+z \} \]
\[ y := x - w \quad \text{Avail} = \{ x-w \} \]
\[ a := w + z \quad \text{Avail} = \{ x-w , w+z \} \]
\[ z := x - w \quad \text{Avail} = \{ x-w \} \]
\[ y := y + z \quad \text{Avail} = \{ x-w \} \]

Computing Available Expressions

The Universe

\[ U = \text{The set of all expressions appearing in the flow graph} \]

Example:

\[ U = \{ a-b, w+x, y*4, x+1, b-c \} \]

\[ E\_GEN \ [B] = \]

The set of expressions computed in the block

\[ x \oplus y \] is included if some statement in B evaluates it

and the block does not assign to \(x\) or \(y\) after that.

\[ E\_KILL \ [B] = \]

The set of expressions that are invalidated because

the block contains an assignment to a variable they use.

\[ E\_IN \ [B] = \]

The set of expressions available at the beginning of block B.

\[ E\_OUT \ [B] = \]

The set of expressions available at the end of block B.
Computing Available Expressions

The Universe
= The set of all expressions appearing in the flow graph

Example: \[ \mathbb{U} = \{ a-b, w+x, y*4, x+1, b-c \} \]

\[ E_{\text{GEN}}[B] = \]

The set of expressions computed in the block
\[ x \oplus y \] is included if some statement in B evaluates it
and the block does not assign to \( x \) or \( y \) after that.

\[ E_{\text{KILL}}[B] = \]

The set of expressions that are invalidated because
the block contains an assignment to a variable they use.

\[ E_{\text{IN}}[B] = \]

The set of expressions available at the beginning of block B.

\[ E_{\text{OUT}}[B] = \]

The set of expressions available at the end of block B.

Recurrence Equations to be Solved:

\[
\begin{align*}
E_{\text{OUT}}[B] & := E_{\text{GEN}}[B] \cup ( E_{\text{IN}}[B] - E_{\text{KILL}}[B] ) \\
E_{\text{IN}}[B] & := \bigcap_{P \text{ is a predecessor of } B} E_{\text{OUT}}[P] \\
E_{\text{IN}}[B_1] & = \{ \} \quad \text{Nothing available before the initial block}
\end{align*}
\]

For \( B \neq B_1 \) (the initial block)
Forward Propagation
(like reaching definitions, but \( \cap \) instead of \( \cup \))

Reaching Definitions
Start with estimates that are too small, and enlarge them.

\[
IN[B] = \bigcup_{p=\text{predecessor}} OUT[P]
\]

Available Expressions
Start with estimates that are too large, and shrink them.

\[
E_{IN}[B] = \bigcap_{p=\text{predecessor}} E_{OUT}[P]
\]

Algorithm to Compute Available Expressions

Input:
E\_GEN and E\_KILL for each block

Output:
E\_IN[B] = Expressions available at beginning of B

Algorithm:

\[
E_{IN}[B_1] := \{}
E_{OUT}[B_1] := E_{GEN}[B_1]
\]

for each block B except B_1 do
\[
E_{OUT}[B] := \bigcup E_{OUT} - E_{KILL}[B]
\]
endfor

while changes occur for any E\_OUT set do

for each block B except B_1 do
\[
E_{IN}[B] := \bigcap_{P \text{ is a predecessor of } B} E_{OUT}[P]
\]
\[
E_{OUT}[B] := E_{GEN}[B] \cup (E_{IN}[B] - E_{KILL}[B])
\]
endfor
endwhile
Conservative, Safe Estimates

- Begin by assuming all expressions available anywhere.
- Work toward a smaller solution.
- If there is a possible definition of $x$ or $y$ then consider $x \oplus y$ as not available.
- We will tend to err by eliminating too many expressions from $E_{IN}$ and $E_{OUT}$.
- Our computed result will be a subset of the expressions that are truly available at point $P$.
- If our computation determines that $x \oplus y$ is available at point $P$, then it surely is.

**We can eliminate its recomputation!**

---

Eliminating Common Global Subexpressions

The Transformation

\[
\begin{align*}
a & := x \oplus y \\
b & := x \oplus y \\
w & := x \oplus y
\end{align*}
\]
Eliminating Common Global Subexpressions

The Transformation

Create a new temporary.

And use it here.
**Eliminating Common Global Subexpressions**

**The Transformation**

\[
\begin{align*}
t & := x \odot y \\
a & := t \\
t & := x \odot y \\
b & := t \\
w & := t
\end{align*}
\]

Copy Propagation may eliminate these statements.

---

**Algorithm**

*Input:* Flow Graph, Available Expression Information  
*Output:* Revised Flow Graph

**Step 1:**
Find a statement such as
\[
w := x \odot y
\]
such that expression \(x \odot y\) is available directly before it.  
[Or: \(x \odot y\) is available in \(E_{IN}[B]\) for the block and there are no assignments to \(x\) or \(y\) before this statement.]
**Algorithm**

**Input:** Flow Graph, Available Expression Information  
**Output:** Revised Flow Graph

**Step 1:**  
Find a statement such as  

\[ w := x \oplus y \]  
such that expression \( x \oplus y \) is available directly before it.  
[Or: \( x \oplus y \) is available in \( E_{IN}[B] \) for the block and there are no assignments to \( x \) or \( y \) before this statement.]

**Step 2:**  
Follow the flow graph edges backward until you hit an evaluation of \( x \oplus y \). Find all such evaluations.

\[ a := x \oplus y \]

\[ b := x \oplus y \]

\[ c := x \oplus y \]

**Step 3:**  
Create a new temporary (say “t”)
Algorithm

**Step 3:**
Create a new temporary (say “t”)

**Step 4:**
Replace all statements found in step 2.

\[
\begin{align*}
&a := x \oplus y \\
&t := x \oplus y \\
&a := t \\

&b := x \oplus y \\
&t := x \oplus y \\
&b := t \\

&c := x \oplus y \\
&t := x \oplus y \\
&c := t
\end{align*}
\]
Algorithm

Step 3:
Create a new temporary (say “t”)

Step 4:
Replace all statements found in step 2.

\[
\begin{align*}
    a &:= x \oplus y \\
    &\downarrow \\
    t &:= x \oplus y \\
    &\downarrow \\
    a &:= t
\end{align*}
\]

\[
\begin{align*}
    b &:= x \oplus y \\
    &\downarrow \\
    t &:= x \oplus y \\
    &\downarrow \\
    b &:= t \\
    c &:= x \oplus y \\
    &\downarrow \\
    t &:= x \oplus y \\
    &\downarrow \\
    c &:= t
\end{align*}
\]

Notes:
- Copy propagation may eliminate some of the extra assignments (but might not)
- Program size could grow
- Want to limit this effect...
  If more than 1 statement found in step 2, just forget it.

Copy Propagation

A copy statement

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

Where do the copies come from:
- IR code generation
- Common Sub-Expression Elimination
- Other Optimizations
Copy Propagation

We can use $y$ instead of $x$ if...

- The only definition of $x$ reaching $a := b \oplus x$ is $x := y$, and
- There is no assignment to $y$ on any path from $x := y$ to $a := b \oplus x$. 

© Harry H. Porter, 2006
Copy Propagation

We can not propagate the copy in this example:

\[
\begin{align*}
    x &:= y \\
    a &:= b \oplus x \\
    y &:= 47
\end{align*}
\]

There must be no assignment to \( y \) on any path from \( x := y \) to \( a := b \oplus x \).

We can use \( y \) instead of \( x \) if...

- The only definition of \( x \) reaching \( a := b \oplus x \) is \( x := y \), and

- There is no assignment to \( y \) on any path from \( x := y \) to \( a := b \oplus x \).
We can use $y$ instead of $x$ if...

- The only definition of $x$ reaching $a := b \oplus x$ is $x := y$, and

  Compute the U-D Chains and use that info to determine this!

- There is no assignment to $y$ on any path from $x := y$ to $a := b \oplus x$.

  A new Data Flow problem!
Look at the entire Control Flow Graph
Identify all copy statements.
Two copy statements are different, even if they have the same variables!

Example:
Universe = ???

\[
\begin{align*}
S_1 &: x := y \\
S_2 &: a := b*3 \\
S_3 &: c := x+1 \\
S_4 &: a := d \\
S_5 &: b := x+b \\
S_6 &: x := y \\
S_7 &: x := a+c \\
S_8 &: z := w \\
S_9 &: c := a-1 \\
\end{align*}
\]
Look at the entire Control Flow Graph
Identify all copy statements.
Two copy statements are different,
even if they have the same variables!

Example:
Universe = { S1: x := y
S4: a := d
S6: x := y
S8: z := w }

A block “kills” a copy
x := y
if it contains an assignment to x or y...
A block “kills” a copy

\[ x := y \]

if it contains an assignment to \( x \) or \( y \)...

... unless the block contains the copy itself and does not assign to \( x \) or \( y \) after the copy.

For each basic block, we first compute...

**C_GEN [B]**

The set of all copy statements in basic block \( B \), not killed before they reach the end of the block.

**C_KILL [B]**

The set of all copies in \( \bigcup \) that are killed by block \( B \).
Then, Use Data Flow to Compute...

\( \text{C\_IN [B]} \)

The set of all copy statements \( x := y \) such that every path from the initial block to the beginning of \( B \) contains the copy and there are no assignments to \( x \) or \( y \) on any path from the copy statement to the beginning of block \( B \).

[Technically, there must be no assignments on the path between the last occurrence of the copy and the beginning of block \( B \).]

\( \text{C\_OUT [B]} \)

Same, at the end of the block.

The Data Flow Equations

\[
\begin{align*}
\text{C\_OUT[B]} & := \text{C\_GEN[B]} \cup ( \text{C\_IN[B]} - \text{C\_KILL[B]} ) \\
\text{C\_IN[B]} & := \bigcap_{P \text{ is a predecessor of B}} \text{C\_OUT[P]} \\
\text{C\_IN[B1]} & = \{\} \quad \text{Nothing available before the initial block}
\end{align*}
\]

For \( B \neq B1 \) (the initial block)
## The Data Flow Equations

\[
C_{OUT}[B] := C_{GEN}[B] \cup (C_{IN}[B] - C_{KILL}[B])
\]

\[
C_{IN}[B] := \bigcap_{P \text{ is a predecessor of } B} C_{OUT}[P] 
\]

For \( B \neq B_1 \) (the initial block)

\[
C_{IN}[B_1] = {} \quad \text{Nothing available before the initial block}
\]

These equations are identical to the Available Expression equations!

---

## Copy Deletion Algorithm

**Input:**
- Control Flow Graph
- U-D Chain info
- D-U Chain info
- Results of Data Flow Analysis; \( C_{IN}[B] \), for each block

**Output:**
- Modified Flow Graph
Copy Deletion Algorithm

for each copy statement C: x:=y do
    Determine the set of all uses of x that are reached by C.
    Call such stmts U1, U2, U3, ... UN.
    for each use Ui: ... := ... x... do
        Let B be the basic block containing Ui.
        if C ∈ C_IN[B] and there are no definitions of x or y prior to Ui within B then
            It might be okay to delete C. Keep checking other uses.
        else
            We must not delete C!
            Skip to the next copy statement.
        endif
    endfor
delete C
modify all uses U1, U2, ... UN
endfor