Reading Assignment

- Read pages 1–30 of “Automatic Program Optimization,” by Ron Cytron. (Linked from the class Web page.)
**Data Flow Frameworks**

- **Data Flow Graph:**
  
  Nodes of the graph are basic blocks or individual instructions.
  
  Arcs represent flow of control.

- **Forward Analysis:**
  
  Information flow is the same direction as control flow.

- **Backward Analysis:**
  
  Information flow is the opposite direction as control flow.

- **Bi-directional Analysis:**
  
  Information flow is in both directions. (Not too common.)
• Meet Lattice

  Represents solution space for the data flow analysis.

  \[
  \begin{array}{c}
  T \\
  \Downarrow \\
  \ldots \\
  \Downarrow \\
  \bot
  \end{array}
  \]

• Meet operation
  (And, Or, Union, Intersection, etc.)

  Combines solutions from predecessors or successors in the control flow graph.
• Transfer Function
  Maps a solution at the top of a node to a solution at the end of the node (forward flow)
  or
  Maps a solution at the end of a node to a solution at the top of the node (backward flow).
**Example: Available Expressions**

This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.
The Meet Lattice is:

T (Expression is Available)

F (Expression is Not Available)

As initial values, at the top of the start node, nothing is available. Hence, for a given expression, 
\[ \text{AvailIn}(b_0) = F \]

We choose an expression, and consider all the variables that contribute to its evaluation. Thus for \( e_1 = a+b-c \), \( a, b \) and \( c \) are \( e_1 \)'s operands.
The transfer function for \( e_1 \) in block \( b \) is defined as:

If \( e_1 \) is computed in \( b \) after any assignments to \( e_1 \)'s operands in \( b \)
Then \( \text{AvailOut}(b) = T \)
Elsif any of \( e_1 \)'s operands are changed after the last computation of \( e_1 \) or \( e_1 \)'s operands are changed without any computation of \( e_1 \)
Then \( \text{AvailOut}(b) = F \)
Else \( \text{AvailOut}(b) = \text{AvailIn}(b) \)

The meet operation (to combine solutions) is:

\[
\text{AvailIn}(b) = \bigwedge_{p \in \text{Pred}(b)} \text{AvailOut}(p)
\]
Example: $e_1 = v + w$

```
v = 9
w = 5
x = v + w
y = v + w
z = v + w
stop
v = 2
F
T
F
T
F
F
T
F
F
T
F
```
Circularities Require Care

Since data flow values can depend on themselves (because of loops), care is required in assigning initial "guesses" to unknown values. Consider

If the flow value on the loop backedge is initially set to false, it can never become true. (Why?) Instead we should use True, the identity for the AND operation.
**Very Busy Expressions**

This is an interesting variant of available expression analysis.

An expression is *very busy* at a point if it is *guaranteed* that the expression will be computed at some time in the future.

Thus starting at the point in question, the expression must be reached before its value changes.

Very busy expression analysis is a backward flow analysis, since it propagates information about future evaluations backward to “earlier” points in the computation.
The meet lattice is:

\[
\begin{align*}
\text{T (Expression is Very Busy)} \\
\text{F (Expression is Not Very Busy)}
\end{align*}
\]

As initial values, at the end of all exit nodes, nothing is very busy. Hence, for a given expression,

\[
\text{VeryBusyOut(b_{last})} = \text{F}
\]
The transfer function for \( e_1 \) in block \( b \) is defined as:

If \( e_1 \) is computed in \( b \) before any of its operands
Then \( \text{VeryBusyIn}(b) = T \)
Elsif any of \( e_1 \)'s operands are changed before \( e_1 \) is computed
Then \( \text{VeryBusyIn}(b) = F \)
Else \( \text{VeryBusyIn}(b) = \text{VeryBusyOut}(b) \)

The meet operation (to combine solutions) is:

\[
\text{VeryBusyOut}(b) = \text{AND} \ \text{VeryBusyIn}(s) \\
\text{\quad} s \in \text{Succ}(b)
\]
Example: $e_1 = v + w$

\[
\begin{align*}
v &= 2 \\
w &= 5
\end{align*}
\]

\[
\begin{align*}
v &= 3 \\
x &= v + w
\end{align*}
\]

\[
\begin{align*}
u &= v + w
\end{align*}
\]

stop
v = 2
w = 5

v = 3
x = v + w

F
T

F
T

F
T

F
T

F
T

F
T

F
T

F
T

stop

F
T

Or here?

Move v+w here?
Identifying Identical Expressions

We can hash expressions, based on hash values assigned to operands and operators. This makes recognizing potentially redundant expressions straightforward.

For example, if \( H(a) = 10 \), \( H(b) = 21 \) and \( H(+) = 5 \), then (using a simple product hash),
\[
H(a+b) = 10 \times 21 \times 5 \; \text{Mod TableSize}
\]
Effects of Aliasing and Calls

When looking for assignments to operands, we must consider the effects of pointers, formal parameters and calls.

An assignment through a pointer (e.g, \( *p = val \)) \textit{kills} all expressions dependent on variables \( p \) might point too. Similarly, an assignment to a formal parameter kills all expressions dependent on variables the formal might be bound to.

A call \textit{kills} all expressions dependent on a variable changeable during the call.

Lacking careful alias analysis, pointers, formal parameters and calls can \textit{kill} all (or most) expressions.
**Very Busy Expressions and Loop Invariants**

Very busy expressions are ideal candidates for invariant loop motion. If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.
for (...) {
    if (...) 
        a=b+c;
    else a=d+c;}

for (...) {
    if (a>b+c) 
        x=1;
    else x=0;}

b+c is not very busy at loop entrance

b+c is very busy at loop entrance
Reaching Definitions

We have seen reaching definition analysis formulated as a set-valued problem. It can also be formulated on a per-definition basis.

That is, we ask “What blocks does a particular definition to v reach?”

This is a boolean-valued, forward flow data flow problem.
Initially, DefIn(b₀) = false.

For basic block b:

\[
\text{DefOut}(b) = \begin{cases} 
\text{True} & \text{if the definition being analyzed is the last definition to } v \text{ in } b \\ 
\text{False} & \text{if any other definition to } v \text{ occurs in } b \\ 
\text{DefIn}(b) & \text{else} 
\end{cases}
\]

The meet operation (to combine solutions) is:

\[
\text{DefIn}(b) = \bigvee_{p \in \text{Pred}(b)} \text{DefOut}(p)
\]

To get all reaching definition, we do a series of single definition analyses.
**Live Variable Analysis**

This is a boolean-valued, backward flow data flow problem.

Initially, LiveOut(b_{last}) = false.

For basic block b:

LiveIn(b) =

- If the variable is used before it is defined in b
  - Then True
- Elsif it is defined before it is used in b
  - Then False
- Else LiveOut(b)

The meet operation (to combine solutions) is:

\[ \text{LiveOut}(b) = \bigvee_{s \in \text{Succ}(b)} \text{LiveIn}(s) \]
**Bit Vectoring Data Flow Problems**

The four data flow problems we have just reviewed all fit within a *single* framework.

Their solution values are Booleans (bits).

The meet operation is And or OR.

The transfer function is of the general form

\[ \text{Out}(b) = (\text{In}(b) - \text{Kill}_b) \cup \text{Gen}_b \]

or

\[ \text{In}(b) = (\text{Out}(b) - \text{Kill}_b) \cup \text{Gen}_b \]

where \( \text{Kill}_b \) is true if a value is “killed” within \( b \) and \( \text{Gen}_b \) is true if a value is “generated” within \( b \).
In Boolean terms:
\[ \text{Out}(b) = (\text{In}(b) \text{ AND } \text{Not } \text{Kill}_b) \text{ OR } \text{Gen}_b \]
or
\[ \text{In}(b) = (\text{Out}(b) \text{ AND } \text{Not } \text{Kill}_b) \text{ OR } \text{Gen}_b \]

An advantage of a bit vectoring data flow problem is that we can do a series of data flow problems “in parallel” using a bit vector.

Hence using ordinary word-level ANDs, ORs, and NOTs, we can solve 32 (or 64) problems simultaneously.
Example

Do live variable analysis for \( u \) and \( v \), using a 2 bit vector:

\[
\begin{align*}
\text{Live}=0,0 & \quad \text{Gen}=0,0 \\
\quad \quad \text{Kill}=0,1 \\
\quad \quad \text{v}=1
\end{align*}
\]

\[
\begin{align*}
\text{Live}=0,1 & \quad \text{Gen}=0,0 \\
\quad \quad \text{Kill}=1,0 \\
\text{u}=0
\end{align*}
\]

\[
\begin{align*}
\text{Live}=1,1 & \quad \text{Gen}=1,0 \\
\quad \quad \text{Kill}=0,0 \\
\quad \quad \text{a}=u
\end{align*}
\]

\[
\begin{align*}
\text{Live}=1,1 & \quad \text{Gen}=1,1 \\
\quad \quad \text{Kill}=0,1 \\
\quad \quad \text{v}=2
\end{align*}
\]

\[
\begin{align*}
\text{Live}=1,0 & \quad \text{Gen}=0,0 \\
\quad \quad \text{Kill}=0,1 \\
\text{print}(u,v)
\end{align*}
\]

We expect no variable to be live at the start of \( b_0 \). (Why?)