Code Optimization
Data flow analysis

➢ Goal:
  - collect information about how a procedure manipulates its data

➢ This information is used in various optimizations
  - For example, knowledge about what expressions are available at some point helps in common subexpression elimination.

➢ IMPORTANT!
  - Data flow analysis should never tell us that a transformation is safe when in fact it is not.
  - It is better to not perform a valid optimization that to perform one that changes the function of the program.
Data flow analysis

➢ IMPORTANT!

- When doing data flow analysis we must be
  - Conservative
    - Do not consider information that may not preserve the behavior of the program
  - Aggressive
    - Try to collect information that is as exact as possible, so we can get the greatest benefit from our optimizations.
Global Iterative Data Flow Analysis

➢ Global:
  - Performed on the flow graph
  - Goal = to collect information at the beginning and end of each basic block

➢ Iterative:
  - Construct data flow equations that describe how information flows through each basic block and solve them by iteratively converging on a solution.
Components of data flow equations

- Sets containing collected information
  - \textbf{in} set: information coming into the BB from outside (following flow of data)
  - \textbf{gen} set: information generated/collected within the BB
  - \textbf{kill} set: information that, due to action within the BB, will affect what has been collected outside the BB
  - \textbf{out} set: information leaving the BB

- Functions (operations on these sets)
  - \textbf{Transfer functions} describe how information changes as it flows through a basic block
  - \textbf{Meet functions} describe how information from multiple paths is combined.
Algorithm sketch

- Typically, a bit vector is used to store the information.
  - For example, in reaching definitions, each bit position corresponds to one definition.
- We use an iterative fixed-point algorithm.
- Depending on the nature of the problem we are solving, we may need to traverse each basic block in a forward (top-down) or backward direction.
  - The order in which we "visit" each BB is not important in terms of algorithm correctness, but is important in terms of efficiency.
- In & Out sets should be initialized in a conservative and aggressive way.

```
Initialize gen and kill sets
Initialize in or out sets (depending on "direction")
while there are no changes in in and out sets {
    for each BB {
        apply meet function
        apply transfer function
    }
}
```
Typical problems

➢ Reaching definitions
  - For each use of a variable, find all definitions that reach it.

➢ Upward exposed uses
  - For each definition of a variable, find all uses that it reaches.

➢ Live variables
  - For a point $p$ and a variable $v$, determine whether $v$ is live at $p$.

➢ Available expressions
  - Find all expressions whose value is available at some point $p$. 
Reaching definitions

➢ Determine which definitions of a variable may reach each use of the variable.
  ■ For each use, list the definitions that reach it. This is also called a ud-chain.
  ■ In global data flow analysis, we collect such information at the endpoints of a basic block, but we can do additional local analysis within each block.

➢ Uses of reaching definitions:
  ■ constant propagation
    ■ we need to know that all the definitions that reach a variable assign it to the same constant
  ■ copy propagation
    ■ we need to know whether a particular copy statement is the only definition that reaches a use.
  ■ code motion
    ■ we need to know whether a computation is loop-invariant
Reaching definitions

➢ A definition $D$ reaches a point $p$ if there is a path from $D$ to $p$ along which $D$ is not killed.

➢ A definition $D$ of a variable $x$ is killed when there is a redefinition of $x$.

➢ How can we represent the set of definitions reaching a point?

- Use a bit string of length $n$, where $n$ is the number of definitions. Set bit $i$ to 1 if definition $i$ reaches that point, set it to 0 otherwise.
Reaching definitions

➢ What is safe?

- To assume that a definition reaches a point even if it turns out not to.
- The computed set of definitions reaching a point \( p \) will be a superset of the actual set of definitions reaching \( p \)
- Goal: make the set of reaching definitions as small as possible (i.e. as close to the actual set as possible)
Reaching definitions

➢ How are the gen and kill sets defined?
  - $\text{gen}[B] = \{\text{definitions that appear in } B \text{ and reach the end of } B\}$
  - $\text{kill}[B] = \{\text{all definitions that never reach the end of } B\}$

➢ What is the direction of the analysis?
  - forward
  - $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$
Reaching definitions

➢ What is the confluence operator?
  ■ union
  ■ \( \text{in}[B] = \bigcup \text{out}[P] \), over the predecessors \( P \) of \( B \)

➢ How do we initialize?
  ■ start small
    ■ Why? Because we want the resulting set to be as small as possible
  ■ for each block \( B \) initialize \( \text{out}[B] = \text{gen}[B] \)
Upward Exposed Uses

➢ Determine what uses of a variable are reached by a specific definition of that variable.

- For each definition, list the uses that are reached by it. This is also called a **du-chain**.
- This is the dual of reaching definitions.
- Useful in instruction scheduling.
- du-chains and ud-chains are different:

```
z > 1
x = 1
z > y
y = x + 1
z = x + 3
print z
```
Upward Exposed Uses

➢ What is the direction of the analysis?
   - backward
   - $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$

➢ How are the $\text{use}$ and $\text{def}$ sets defined?
   - $\text{use}[B] = \{(s,x) \mid s \text{ is a use of } x \text{ in } B \text{ and there is no definition of } x$
     between the beginning of $B$ and $s\}$
   - $\text{def}[B] = \{(s,x) \mid s \text{ is a use of } x \text{ not in } B \text{ and } B \text{ contains a definition of } x\}$

➢ What is the confluence operator?
   - union
   - $\text{out}[B] = \bigcup \text{in}[S]$, over the successors $S$ of $B$
Upward Exposed Uses

➢ How do we initialize?
  ▪ Start small
  ▪ for each block $B$ initialize $\text{in}[B] = \emptyset$

➢ du- and ud- chains are useful in register allocation.
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
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).
How are the **gen** and **kill** sets defined?
- \( \text{gen}[B] = \{ \text{expressions evaluated in B without subsequently redefining its operands} \} \)
- \( \text{kill}[B] = \{ \text{expressions whose operands are redefined in B without reevaluating the expression afterwards} \} \)

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 large
   - for the first block $B_1$ initialize $\text{out}[B_1] = \text{gen}[B_1]$
   - for each block $B$ initialize $\text{out}[B] = \text{U-kil}[B]$
Live variables

➢ Determine whether a given variable is used along a path from a given point to the exit.

➢ A variable $x$ is *live at point $p$* if there is a path from $p$ to the exit along which the value of $x$ is used before it is redefined.

➢ Otherwise, the variable is dead at that point.

➢ Used in:
  - register allocation
  - dead code elimination
Live variables

What is safe?

- To assume that a variable is live at some point even if it may not be.
- The computed set of live variables at point $p$ will be a superset of the actual set of live variables at $p$.
- The computed set of dead variables at point $p$ will be a subset of the actual set of dead variables at $p$.
- Goal: make the set of live variables as small as possible (i.e. as close to the actual set as possible).
Live variables

➢ How are the def and use sets defined?

- \( \text{def}[^B] = \{ \text{variables defined in } \text{B before being used} \} \)
  /* kill */

- \( \text{use}[^B] = \{ \text{variables used in } \text{B before being defined} \} \)
  /* gen */

➢ What is the direction of the analysis?

- backward

- \( \text{in}[^B] = \text{use}[^B] \cup (\text{out}[^B] - \text{def}[^B]) \)}
Live variables

➤ What is the confluence operator?

- union
- $\text{out}[B] = \bigcup \text{in}[S]$, over the successors $S$ of $B$

➤ How do we initialize?

- start small
- for each block $B$ initialize $\text{in}[B] = \emptyset$ or $\text{in}[B] = \text{use}[B]$
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 \( p \).

- Does this make the generated code faster?
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?
   - **gen**\([B]\) = \{all expressions evaluated in \(B\) before any definitions of their operands\}
   - **kill**\([B]\) = \{all expressions whose operands are defined in \(B\) before any possible re-evaluation\}

➢ What is the direction of the analysis?
   - backward
   - **in**\([B]\) = **gen**\([B]\) \(\cup\) (**out**\([B]\) - **kill**\([B]\))
What is the confluence operator?
- intersection
- $\text{out}[B] = \bigcap \text{in}[S]$, over the successors $S$ of $B$

How do we initialize?
- start large
- for each block $B$ initialize $\text{out}[B] = U$
### General framework

<table>
<thead>
<tr>
<th></th>
<th>as small as possible</th>
<th>as large as possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>desired set</td>
<td>as small as possible</td>
<td>as large as possible</td>
</tr>
<tr>
<td>resulting set</td>
<td>larger than actual</td>
<td>smaller than actual</td>
</tr>
<tr>
<td>gen</td>
<td>everything that <em>may</em> be true</td>
<td>everything that <em>must</em> be true</td>
</tr>
<tr>
<td>kill</td>
<td>everything that <em>must</em> be false</td>
<td>everything that <em>may</em> be false</td>
</tr>
<tr>
<td>confluence</td>
<td>union</td>
<td>intersection</td>
</tr>
<tr>
<td>initial</td>
<td>start with small initial set</td>
<td>start with large initial set</td>
</tr>
<tr>
<td>example</td>
<td>live variables (bwd)</td>
<td>very busy expressions (bwd)</td>
</tr>
<tr>
<td>example</td>
<td>reaching definitions (fwd)</td>
<td>available expressions (fwd)</td>
</tr>
</tbody>
</table>
Dataflow analysis example

```
entry

a = 1
b = 2

c = a + b
d = c - a

b = b * d

d = a + b
e = e + 1

b = a + b
e = c - a

a = b * d
b = a - d

exit
```