Lecture 20

COMPILER DESIGN
Announcements

• **HW5: OAT v. 2.0**
  – Records, function pointers, type checking, array-bounds checks, etc.
OTHER DATAFLOW ANALYSES
Iterative Dataflow Analysis

• Find a solution to those constraints by starting from a rough guess
• Start with: $\text{in}[n] = \text{out}[n] = \emptyset$
• They don’t satisfy the constraints
  – $\text{in}[n] \supseteq \text{use}[n]$
  – $\text{in}[n] \supseteq \text{out}[n] \setminus \text{def}[n]$
  – $\text{out}[n] \supseteq \text{in}[n']$ if $n' \in \text{succ}[n]$

• Idea: iteratively re-compute $\text{in}[n]$ & $\text{out}[n]$ where forced to by constraints
  – Each iteration will add variables to the sets $\text{in}[n]$ & $\text{out}[n]$
    (i.e. the live variable sets will increase monotonically)
• We stop when $\text{in}[n]$ & $\text{out}[n]$ satisfy these equations
  (which are derived from the constraints above)
  – $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$
  – $\text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
Generalizing Dataflow Analyses

• This type of iterative analysis for liveness also applies to other analyses
  – Reaching definitions analysis
  – Available expressions analysis
  – Alias Analysis
  – Constant Propagation
  – These analyses follow the same 3-step approach as for liveness

• We’ll see how they are instances of the same kind of algorithm

• The examples work over a canonical IR called *quadruples*
  – Allows easy definition of def[n] and use[n]
  – Slightly “looser” LLVM IR variant that doesn’t require SSA
    i.e., it has *mutable* local variables
  – We will use LLVM-IR-like syntax
## Def / Use for SSA

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = op b c</td>
<td>{a}</td>
<td>{b,c}</td>
<td>arithmetic</td>
</tr>
<tr>
<td>a = load b</td>
<td>{a}</td>
<td>{b}</td>
<td>load</td>
</tr>
<tr>
<td>store c, b</td>
<td>Ø</td>
<td>{b}</td>
<td>store</td>
</tr>
<tr>
<td>a = alloc a t</td>
<td>{a}</td>
<td>Ø</td>
<td>alloc</td>
</tr>
<tr>
<td>a = bitcast b to u</td>
<td>{a}</td>
<td>{b}</td>
<td>bitcast</td>
</tr>
<tr>
<td>a = gep b [c,d,...]</td>
<td>{a}</td>
<td>{b,c,d,...}</td>
<td>getelementptr</td>
</tr>
<tr>
<td>a = f(b₁,...,bₙ)</td>
<td>{a}</td>
<td>{b₁,...,bₙ}</td>
<td>call w/return</td>
</tr>
<tr>
<td>f(b₁,...,bₙ)</td>
<td>Ø</td>
<td>{b₁,...,bₙ}</td>
<td>void call (no return)</td>
</tr>
</tbody>
</table>

### Terminators

<table>
<thead>
<tr>
<th>Terminator</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>br L</td>
<td>Ø</td>
<td>Ø</td>
<td>jump</td>
</tr>
<tr>
<td>br a L₁ L₂</td>
<td>Ø</td>
<td>{a}</td>
<td>conditional branch</td>
</tr>
<tr>
<td>return a</td>
<td>Ø</td>
<td>{a}</td>
<td>return</td>
</tr>
</tbody>
</table>
REACHING DEFINITIONS
• Q: What variable definitions reach a particular use of the variable?

• This analysis is used for constant propagation & copy propagation
  – Constant propagation: If only one definition reaches a particular use, can replace use by the definition
  – Copy propagation: additionally requires that the copied value still has its same value – computed using an available expressions analysis (next)

• Input: Quadruple CFG
• Output: in[n] (resp. out[n]) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n
Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]

\[ \text{out}[1]: \{1\} \]
\[ \text{in}[2]: \{1\} \]
\[ \text{out}[2]: \{2,3\} \]
\[ \text{in}[3]: \{1,2\} \]
\[ \text{out}[3]: \{2,3\} \]
\[ \text{in}[4]: \{2,3\} \]
Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let $\text{defs}[a]$ be the set of nodes that define the variable $a$
- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows

<table>
<thead>
<tr>
<th>Quadruple forms $n$</th>
<th>$\text{gen}[n]$</th>
<th>$\text{kill}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \text{ op } c$</td>
<td>${n}$</td>
<td>$\text{defs}[a] \setminus {n}$</td>
</tr>
<tr>
<td>$a = \text{load } b$</td>
<td>${n}$</td>
<td>$\text{defs}[a] \setminus {n}$</td>
</tr>
<tr>
<td>$\text{store } b, a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1, \ldots, b_n)$</td>
<td>${n}$</td>
<td>$\text{defs}[a] \setminus {n}$</td>
</tr>
<tr>
<td>$f(b_1, \ldots, b_n)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br } L$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br } a \text{ L1 L2}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>return $a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
• Define the constraints that a reaching definitions solution must satisfy

• \( \text{out}[n] \supseteq \text{gen}[n] \)
  “Definitions reaching the end of a node at least include the definitions generated by the node”

• \( \text{in}[n] \supseteq \text{out}[n'] \) if \( n' \) is in \( \text{pred}[n] \)
  “Definitions reaching the beginning of a node include those that reach the exit of any predecessor”

• \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  “Definitions coming into a node \( n \) either reach the end of \( n \) or are killed by it”
  – Equivalently: \( \text{out}[n] \supseteq \text{in}[n] \setminus \text{kill}[n] \)
Reaching Definitions Step 3

• Convert constraints to iterated update equations
  \[ \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \]
  \[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] \setminus \text{kill}[n]) \]

• Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to \( \emptyset \)
  – Iterate the update equations until a fixed point is reached

• Algorithm **terminates** since \( \text{in}[n] \) & \( \text{out}[n] \) increase only *monotonically*
  – At most to a maximum set that includes all variables in the program

• It is **precise** since it finds the *smallest* sets that satisfy the constraints
AVAILABLE EXPRESSIONS
Available Expressions

• Idea: want to perform common subexpression elimination (CSE)

\[
a = x + 1 \quad a = x + 1 \\
\ldots \quad \ldots \\
b = x + 1 \quad b = a
\]

• It is safe if \(x+1\) computes the same value at both places
  – “\(x+1\)” is an available expression

• Dataflow values
  \(\text{in}[n] = \text{set of nodes whose values are available on entry to } n\)
  \(\text{out}[n] = \text{set of nodes whose values are available on exit of } n\)
Available Expressions Step 1

- Define the sets of values
- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows

<table>
<thead>
<tr>
<th>Quadruple forms $n$</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \oplus c$</td>
<td>${n} \setminus \text{kill}[n]$</td>
<td>uses[a]</td>
</tr>
<tr>
<td>$a = \text{load} \ b$</td>
<td>${n} \setminus \text{kill}[n]$</td>
<td>uses[a]</td>
</tr>
<tr>
<td>store $b$, $a$</td>
<td>$\emptyset$</td>
<td>uses[ [x] ]</td>
</tr>
<tr>
<td>$\text{br } L$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br } a \ L1 \ L2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1, \ldots, b_n)$</td>
<td>$\emptyset$</td>
<td>uses[a]$\cup$ uses[ [x] ]</td>
</tr>
<tr>
<td>$f(b_1, \ldots, b_n)$</td>
<td>$\emptyset$</td>
<td>uses[ [x] ]</td>
</tr>
<tr>
<td>return $a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Note the need for “may alias” information...

Note that functions are assumed to be impure
Available Expressions Step 2

• Define constraints that an available expressions solution must satisfy

• \( \text{out}[n] \supseteq \text{gen}[n] \)
  “Expressions made available by \( n \) that reach the end of the node”

• \( \text{in}[n] \subseteq \text{out}[n'] \) if \( n' \) is in \( \text{pred}[n] \)
  “Expressions available at the beginning of a node include those that reach the exit of every predecessor”

• \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  “Expressions available on entry either reach the end of the node or are killed”
  – Equivalently: \( \text{out}[n] \supseteq \text{in}[n] \setminus \text{kill}[n] \)

Note similarities and differences with constraints for “reaching definitions”.
Available Expressions Step 3

• Convert constraints to iterated update equations

\[ in[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \]
\[ \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] \setminus \text{kill}[n]) \]

• Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to the set of all nodes
  – Iterate the update equations until a fixed point is reached

• Algorithm terminates since \( \text{in}[n] \) & \( \text{out}[n] \) decrease only monotonically
  – At most to a minimum of the empty set

• It is precise since it finds the largest sets that satisfy the constraints
GENERAL DATAFLOW ANALYSIS
Comparing Dataflow Analyses

- Look at the update equations in the inner loop of the analyses
- Liveness: \textit{(backward, may)}
  - Let \( \text{gen}[n] = \text{use}[n] \) and \( \text{kill}[n] = \text{def}[n] \)
  - \( \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
  - \( \text{in}[n] := \text{gen}[n] \cup (\text{out}[n] \setminus \text{kill}[n]) \)

- Reaching Definitions: \textit{(forward, may)}
  - \( \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
  - \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] \setminus \text{kill}[n]) \)

- Available Expressions: \textit{(forward, must)}
  - \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
  - \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] \setminus \text{kill}[n]) \)
Very Busy Expressions

- Expression $e$ is *very busy* at location $p$ if every path from $p$ must evaluate $e$ before any variable in $e$ is redefined.

- Optimization: hoisting expressions

- A must-analysis
- A backward analysis

- Dataflow equations?
<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>
The Literature

• Vast literature on dataflow analyses

• 90+% can be described by
  – Forward or backward
  – May or must

• Some oddballs, but not many
  – Bidirectional analyses
Common Features

- All of these analyses have a *domain* over which they solve constraints
  - Liveness, the domain is sets of variables
  - Reaching defns., Available exprs. the domain is sets of nodes

- Each analysis has a notion of gen[n] and kill[n]
  - Used to explain how information propagates across a node

- Each analysis propagates information either *forward* or *backward*
  - **Forward**: in[n] defined in terms of predecessor nodes’ out[]
  - **Backward**: out[n] defined in terms of successor nodes’ in[]

- Each analysis has a way of aggregating information
  - Liveness & reaching definitions take union (U)
  - Available expressions uses intersection (∩)
  - Union expresses a property that holds for *some* path (existential) -- *may*
  - Intersection expresses a property that holds for *all* paths (universal) -- *must*
A forward dataflow analysis can be characterized by

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then “$x \in \ell$” means “$x$ has the property”

2. For each node $n$, a flow function $F_n : \mathcal{L} \to \mathcal{L}$
   - So far we’ve seen $F_n(\ell) = \text{gen}[n] \cup (\ell \setminus \text{kill}[n])$
   - So: $\text{out}[n] = F_n(\text{in}[n])$
   - “If $\ell$ is a property that holds before the node $n$, then $F_n(\ell)$ holds after $n$”

3. A combining operator $\Pi$
   - “If we know either $\ell_1$ or $\ell_2$ holds on entry to node $n$, we know at most $\ell_1 \Pi \ell_2$”
   - $\text{in}[n] := \prod_{n' \in \text{pred}[n]} \text{out}[n']$
Generic Iterative (Forward) Analysis

for all \( n \), \( \text{in}[n] := \top \), \( \text{out}[n] := \top \)
repeat until no change
  for all \( n \)
    \( \text{in}[n] := \prod_{n' \in \text{pred}[n]} \text{out}[n'] \)
    \( \text{out}[n] := F_n(\text{in}[n]) \)
  end
end

\( \top \in \mathcal{L} \) (“top”) represents having the “maximum” amount of information
  Having “more” information enables more optimizations
  “Maximum” amount could be inconsistent with the constraints
  Iteration refines the answer, eliminating inconsistencies
Structure of $\mathcal{L}$

- The domain has structure that reflects the “amount” of information contained in each dataflow value
- Some dataflow values are more informative than others
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever $\ell_2$ provides at least as much information as $\ell_1$
  - The dataflow value $\ell_2$ is “better” for enabling optimizations

- Example 1: liveness analysis --- smaller sets of variables more informative
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments
  - So: $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \supseteq \ell_2$

- Example 2: available expressions --- larger sets of nodes more informative
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination
  - So: $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \subseteq \ell_2$
\( L \) as a Partial Order

- \( L \) is a *partial order* defined by the ordering relation \( \sqsubseteq \)
- A partial order is an ordered set
- Some of the elements might be *incomparable*
  - That is, there might be \( \ell_1, \ell_2 \in L \) such that neither \( \ell_1 \sqsubseteq \ell_2 \) nor \( \ell_2 \sqsubseteq \ell_1 \)

- Properties of a partial order
  - *Reflexivity*: \( \ell \sqsubseteq \ell \)
  - *Transitivity*: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_3 \) implies \( \ell_1 \sqsubseteq \ell_2 \)
  - *Anti-symmetry*: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_1 \) implies \( \ell_1 = \ell_2 \)

- Examples
  - Integers ordered by \( \leq \)
  - Types ordered by \( < \)
  - Sets ordered by \( \subseteq \) or \( \supseteq \)
Subsets of \{a, b, c\} ordered by \(\subseteq\)

Partial order presented as a Hasse diagram

Height is 3

order \(\sqsubseteq\) is \(\subseteq\)  
meet \(\sqcap\) is \(\cap\)  
join \(\sqcup\) is \(\cup\)
The combining operator $\cap$ is called the “meet” operation.

It constructs the greatest lower bound:

- $\ell_1 \cap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \cap \ell_2 \sqsubseteq \ell_2$
  - “the meet is a lower bound”
- If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \cap \ell_2$
  - “there is no greater lower bound”

Dually, the $\cup$ operator is called the “join” operation.

It constructs the least upper bound:

- $\ell_1 \sqsubseteq \ell_1 \cup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \cup \ell_2$
  - “the join is an upper bound”
- If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \cup \ell_2 \sqsubseteq \ell$
  - “there is no smaller upper bound”

A partial order that has all meets and joins is called a lattice.

- If it has just meets, it’s called a meet semi-lattice.
Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n)
  \[ \text{out}[n] := F_n(\text{in}[n]) \]

- Equivalently:
  \[ \text{out}[n] := F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n']) \]
  - By definition of \text{in}[n]

- We can write this as a simultaneous update of the vector of out[n]
  - let \( x_n = \text{out}[n] \)
  - Let \( X = (x_1, x_2, \ldots, x_n) \) it’s a vector of points in \( L \)
  - \( F(X) = (F_1(\prod_{j \in \text{pred}[1]} \text{out}[j]), F_2(\prod_{j \in \text{pred}[2]} \text{out}[j]), \ldots, F_n(\prod_{j \in \text{pred}[n]} \text{out}[j])) \)

- Any solution to the constraints is a fixpoint \( X \) of \( F \), i.e., \( F(X) = X \)
Iteration Computes Fixpoints

• Let $X_0 = (T, T, \ldots, T)$

• Each loop through the algorithm apply $F$ to the old vector
  $X_1 = F(X_0)$
  $X_2 = F(X_1)$
  …

• $F^{k+1}(X) = F(F^k(X))$
• A fixpoint is reached when $F^k(X) = F^{k+1}(X)$
  – That’s when the algorithm stops

• Wanted: a maximal fixpoint
  – Because that one is more informative/useful for performing optimizations
Monotonicity & Termination

- Each flow function $F_n$ maps lattice elements to lattice elements.

- To be sensible, it should be monotonic. 
  $F : \mathcal{L} \rightarrow \mathcal{L}$ is monotonic iff $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
  - Intuitively: “If you have more information entering a node, you have more information leaving the node”

- Monotonicity lifts point-wise to the function: $F : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  - Vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each $i$

- Note that $F$ is consistent: $F(X_0) \sqsubseteq X_0$
  - So each iteration moves at least one step down the lattice
    
    \[
    \ldots \sqsubseteq F(F(X_0)) \sqsubseteq F(X_0) \sqsubseteq X_0
    \]

- Therefore, # of steps needed to reach a fixpoint is at most (height $H$ of $\mathcal{L}$) * (#nodes): $O(Hn)$
Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise*
  - If $\mathcal{L}$ is a lattice, then so is \{ $f : X \rightarrow \mathcal{L}$ \} where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$

- Like *types*, the dataflow lattices are *static approx.* to dynamic behavior
  - Could pick a lattice based on subtyping
    - Or other information

- Points in the lattice are sometimes called dataflow “*facts*”
## One Cut at the Dataflow Design Space

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>
“Classic” Constant Propagation

• Constant propagation can be formulated as a dataflow analysis

• Idea: propagate and fold integer constants in one pass
  \[ x = 1; \quad \quad x = 1; \]
  \[ y = 5 + x; \quad y = 6; \]
  \[ z = y \times y; \quad z = 36; \]

• Information about a single variable
  – Variable is never defined
  – Variable has a single, constant value
  – Variable is assigned multiple values
Domains for Constant Propagation

- We can make a constant propagation lattice $\mathcal{L}$ for one variable like

  \[
  \top = \text{multiple values}
  \]

  \[
  \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots
  \]

  \[
  \bot = \text{never defined}
  \]

- To accommodate multiple variables, we take the product lattice, with one element per variable
  - Assuming there are three variables, $x$, $y$, and $z$, the elements of the product lattice are of the form $(\ell_x, \ell_y, \ell_z)$
  - Alternatively, think of the product domain as a context that maps variable names to their “abstract interpretations”

- What are “meet” and “join” in this product lattice?
- What is the height of the product lattice?
Flow Functions

• Consider the node \( x = y \text{ op } z \)

• \( F(\ell_x, \ell_y, \ell_z) = ? \)

\[
\begin{align*}
F(\ell_x, T, \ell_z) &= (T, T, \ell_z) \quad \text{“If either input might have multiple values the result of the operation might too.”} \\
F(\ell_x, \ell_y, T) &= (T, \ell_y, T) \\
\end{align*}
\]

\[
\begin{align*}
F(\ell_x, \bot, \ell_z) &= (\bot, \bot, \ell_z) \quad \text{“If either input is undefined the result of the operation is too.”} \\
F(\ell_x, \ell_y, \bot) &= (\bot, \ell_y, \bot) \\
\end{align*}
\]

• \( F(\ell_x, i, j) = (i \text{ op } j, i, j) \quad \text{“If the inputs are known constants, calculate the output statically.”} \)

• Flow functions for the other nodes are easy

• Monotonic?

• Distributes over meets?
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS
Best Possible Solution

• Suppose we have a CFG

• If \( \exists \) path \( p_1 \) starting from the root node (function entry) traversing the nodes \( n_0, n_1, n_2, \ldots, n_k \)

• The best possible information along \( p_1 \)
  \[ \ell_{p_1} = F_{n_k}(\ldots F_{n_2}(F_{n_1}(F_{n_0}(T)))\ldots) \]

• Best solution at the output is some \( \ell \sqsubseteq \ell_p \) for all paths \( p \)

• Meet-over-paths (MOP) solution
  \[ \bigcap_{p \in \text{paths to } n} \ell_p \]

Best answer here is
\[ F_5(F_3(F_2(F_1(T)))) \bigcap F_5(F_4(F_2(F_1(T)))) \]
What about quality of iterative solutions?

• Does the iterative solution compute the MOP solution?

\[ \text{out}[n] = F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n']) \text{ vs. } \prod_{p \in \text{paths_to}[n]} \ell_p? \]

• Answer: Yes, if the flow functions distribute over \( \prod \)
  
  – Distributive means: \( \prod_i F_n(\ell_i) = F_n(\prod_i \ell_i) \)
  
  – Proof is a bit tricky & beyond the scope of this class
  
  – Difficulty: loops in the CFG may mean there are infinitely many paths

• Not all analyses give MOP solution
  
  – They are more conservative
Reaching Definitions is MOP

- $F_n[x] = \text{gen}[n] \cup (x \setminus \text{kill}[n])$

- Does $F_n$ distribute over meet $\sqcap = \cup$?

- $F_n[x \sqcap y]$
  - $= \text{gen}[n] \cup ((x \cup y) \setminus \text{kill}[n])$
  - $= \text{gen}[n] \cup ((x \setminus \text{kill}[n]) \cup (y \setminus \text{kill}[n]))$
  - $= (\text{gen}[n] \cup (x \setminus \text{kill}[n])) \cup (\text{gen}[n] \cup (y \setminus \text{kill}[n]))$
  - $= F_n[x] \cup F_n[y]$
  - $= F_n[x] \sqcap F_n[y]$

- Therefore, Reaching Definitions with iterative analysis always terminates with the MOP (i.e., best) solution

- In fact, the other three analyses (i.e., liveness, available expressions, & very busy expressions) are all MOP
Constprop Iterative Solution

\[
\text{if } x > 0 \\
\begin{align*}
y &= 1 \\
z &= 2
\end{align*}
\begin{align*}
y &= 2 \\
z &= 1
\end{align*}
\]

\[
(x = y + z) = (T, T, T) \text{ iterative solution}
\]
MOP Solution ≠ Iterative Solution

\[
\begin{align*}
(\bot, \bot, \bot) & \quad \text{if } x > 0 \\
(\bot, \bot, \bot) & \quad y = 1 \quad \text{y = 2} \\
(\bot, 1, \bot) & \quad z = 2 \quad z = 1 \\
(\bot, 1, 2) & \quad x = y + z \\
\end{align*}
\]

MOP solution \( (3, 1, 2) \sqcap (3, 2, 1) = (3, T, T) \)
What Problems are Distributive?

• Many analyses of program structure are distributive
  – Liveness Analysis
  – Available Expressions
  – Reaching Definitions
  – Very Busy Expressions

• These express properties on how the program computes
What Problems are Not Distributive?

- Analyses of *what* the program computes
  - The output is a constant, positive, and so on
  - Constprop is an example as we have just seen
Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graphs without loops)

- $O(n)$ nodes
- $O(n)$ edges
- $O(2^n)$ paths*
  - At each branch there is a choice of 2 directions

* Incidentally, a similar idea can be used to force ML / Haskell type inference to need to construct a type that is exponentially big in the size of the program!
Review of (& Additional) Terminology

Review
• Must vs. May
• Forward vs. Backward
• Distributive vs. non-Distributive

Additional
• Flow-sensitive vs. Flow-insensitive
• Context-sensitive vs. Context-insensitive
• Path-sensitive vs. Path-insensitive
Dataflow Analysis: Summary

• Many dataflow analyses fit into a common framework
• Key idea: *Iterative solution* of a system of equations over a *lattice*
  – Iteration terminates if flow functions are monotonic
  – Equivalent to the MOP answer if flow functions distribute over meet ($\sqcap$)

• Dataflow analyses as presented work for an “imperative” IR
  – Values of temporary variables are updated (“mutated”) during evaluation
  – Such mutation complicates calculations
  – SSA = “Single Static Assignment” eliminates this problem
    • By introducing more temporaries --- each one assigned to only once

• Next: Converting to SSA, finding loops and dominators in CFGs