Lecture 24

CIS 341: COMPILERS
Announcements

• HW6: Dataflow Analysis
  – Due: Weds. April 26th

  NOTE: See Piazza for an update…
  TLDR: "simple" regalloc should not suffice.
  Change gradedtests.ml \(\geq\) to >

• FINAL EXAM: Thursday, May 4th noon – 2:00p.m.
OTHER DATAFLOW ANALYSES
Generalizing Dataflow Analyses

• The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  – Reaching definitions analysis
  – Available expressions analysis
  – Alias Analysis
  – Constant Propagation
  – These analyses follow the same 3-step approach as for liveness.

• To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called quadruples
  – Allows easy definition of def[n] and use[n]
  – A “looser” variant of LLVM’s IR that doesn’t require the “static single assignment” – i.e. it has mutable local variables
Quadruple Format

- A Quadruple sequence is just a control-flow graph (flowgraph) where each node is a quadruple:

- Quadruple forms n:
  - def[n] use[n] description
  
a = b op c {a} {b,c} arithmetic
  
a = load b {a} {b} load
  
store a := b Ø {b} store
  
a = f(b_1,...,b_n) {a} {b_1,...,b_n} call w/return
  
f(b_1,...,b_n) Ø {b_1,...,b_n} call no return

  
br L Ø Ø jump
  
br a L1 L2 Ø {a} branch
  
return a Ø {a} return
REACHING DEFINITIONS
Reaching Definition Analysis

• Question: what uses in a program does a given variable definition reach?

• This analysis is used for constant propagation & copy prop.
  – If only one definition reaches a particular use, can replace use by the definition (for constant propagation).
  – 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

out[1]: {1}
in[2]:    {1}
```

```
c = b * b

out[2]: {1,2}
in[3]:    {1,2}
```

```
b = c + 1

out[3]: {2,3}
in[4]:    {2,3}
```

```
return b * a
```

Note how SSA simplifies this analysis:
- each uid already uniquely names a node
- the "kill" information is unnecessary
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>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$a = \text{load } b$</td>
<td>{n}</td>
<td>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$\text{store } a := b$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1, ..., b_n)$</td>
<td>{n}</td>
<td>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$f(b_1, ..., b_n)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br } L$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{br a } L1 \ L2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$L:$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{return } a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy.

- \( \text{out}[n] \supseteq \text{gen}[n] \)
  
  “The definitions that reach 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 \( \text{pred}[n] \)
  
  “The definitions that reach the beginning of a node include those that reach the exit of any predecessor”

- \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  
  “The definitions that come in to a node either reach the end of the node or are killed by it.”
  
  - Equivalently: \( \text{out}[n] \supseteq \text{in}[n] - \text{kill}[n] \)
Reaching Definitions Step 3

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

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

- The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) increase only monotonically
  - At most to a maximum set that includes all variables in the program

- The algorithm is precise because it finds the smallest sets that satisfy the constraints.
AVAILABLE EXPRESSIONS
Available Expressions

- Idea: want to perform common subexpression elimination:
  - \( a = x + 1 \quad a = x + 1 \)
  - \( \ldots \quad \ldots \)
  - \( b = x + 1 \quad b = a \)

- This transformation is safe if \( x+1 \) means computes the same value at both places (i.e. \( x \) hasn’t been assigned).
  - “\( x+1 \)” is an available expression

- Dataflow values:
  - \( \text{in}[n] = \) set of nodes whose values are available on entry to \( n \)
  - \( \text{out}[n] = \) 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:
- Quadruple forms \( n \):

  \[
  \begin{align*}
  a &= b \text{ op } c & \text{gen}[n] &= \{n\} - \text{kill}[n] & \text{kill}[n] &= \text{uses}[a] \\
  a &= \text{load } b & \text{gen}[n] &= \{n\} - \text{kill}[n] & \text{kill}[n] &= \text{uses}[a] \\
  \text{store } a &:= b & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \text{uses}[x] \\
  \text{br } L & & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \emptyset \\
  \text{br } a \ L1 \ L2 & & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \emptyset \\
  L: & & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \emptyset \\
  a &= f(b_1, \ldots, b_n) & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \text{uses}[a] \cup \text{uses}[x] \\
  f(b_1, \ldots, b_n) & & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \text{uses}[x] \\
  \text{return } a & & \text{gen}[n] &= \emptyset & \text{kill}[n] &= \emptyset
  \end{align*}
  \]

- Note the need for “may alias” information...
- Note that functions are assumed to be impure...
Available Expressions Step 2

- Define the constraints that an available expressions solution must satisfy.
- \(\text{out}[n] \supseteq \text{gen}[n]\)
  
  “The expressions made available by \(n\) that reach the end of the node”

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

- \(\text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n]\)
  
  “The expressions available on entry either reach the end of the node or are killed by it.”
  - Equivalently: \(\text{out}[n] \supseteq \text{in}[n] - \text{kill}[n]\)

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

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

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

• The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) decrease only monotonically
  – At most to a minimum of the empty set

• The algorithm is precise because 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: (backward)
    – Let \( \text{gen}[n] = \text{use}[n] \) and \( \text{kill}[n] = \text{def}[n] \)
    – \( \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
    – \( \text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \)

  • Reaching Definitions: (forward)
    – \( \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
    – \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

  • Available Expressions: (forward)
    – \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
    – \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)
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 is 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)
  - Intersection expresses a property that holds for *all* paths (universal)
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 $F_n(\ell) = \text{gen}[n] \cup (\ell - \text{kill}[n])$
   - 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 $\sqcap$
   - “If we know either $\ell_1$ or $\ell_2$ holds on entry to node $n$, we know at most $\ell_1 \sqcap \ell_2$”
   - $\text{in}[n] := \sqcap_{n' \in \text{pred}[n]} \text{out}[n']$
Generic Iterative (Forward) Analysis

for all $n$, $\text{in}[n] := T$, $\text{out}[n] := T$
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

• Here, $T \in L$ (“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 $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ whenever $\mathcal{L}_2$ provides at least as much information as $\mathcal{L}_1$.
  - The dataflow value $\mathcal{L}_2$ is “better” for enabling optimizations.

- Example 1: for liveness analysis, smaller sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \supseteq \mathcal{L}_2$

- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \subseteq \mathcal{L}_2$
\( \mathcal{L} \) as a Partial Order

- \( \mathcal{L} \) 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 \( \ell_1, \ell_2 \in \mathcal{L} \) 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 \(\cup\)
Meets and Joins

- The combining operator $\sqcap$ is called the “meet” operation.
- It constructs the greatest lower bound:
  - $\ell_1 \sqcap \ell_2 \subseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \subseteq \ell_2$
    “the meet is a lower bound”
  - If $\ell \subseteq \ell_1$ and $\ell \subseteq \ell_2$ then $\ell \subseteq \ell_1 \sqcap \ell_2$
    “there is no greater lower bound”

- Dually, the $\sqcup$ operator is called the “join” operation.
- It constructs the least upper bound:
  - $\ell_1 \subseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \subseteq \ell_1 \sqcup \ell_2$
    “the join is an upper bound”
  - If $\ell_1 \subseteq \ell$ and $\ell_2 \subseteq \ell$ then $\ell_1 \sqcup \ell_2 \subseteq \ell$
    “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.
• Information about individual nodes or variables can be lifted pointwise:
  – If $L$ is a lattice, then so is $\{ f : X \rightarrow L \}$ 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 approximations to the dynamic behavior:
  – Could pick a lattice based on subtyping:
    – Or other information:

• Points in the lattice are sometimes called dataflow “facts”
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 $\text{out}[n]$ values:
  
  – let $x_n = \text{out}[n]$
  
  – Let $X = (x_1, x_2, \ldots, x_n)$ it’s a vector of points in $\mathcal{L}$

  – $F(X) = (F_1(\prod_{j \in \text{pred}[1]} \text{out}[j]), F_2(\prod_{j \in \text{pred}[2]} \text{out}[j]), \ldots, F_n(\prod_{j \in \text{pred}[n]} \text{out}[j]))$

• Any solution to the constraints is a fixpoint $X$ of $F$
  
  – i.e. $F(X) = X$
• 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 is should be *monotonic*:

  $F : \mathcal{L} \rightarrow \mathcal{L}$ is monotonic iff:
  $\ell_1 \subseteq \ell_2$ implies that $F(\ell_1) \subseteq F(\ell_2)$
  – Intuitively: “If you have more information entering a node, then you have more information leaving the node.”

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

• Note that $F$ is consistent: $F(X_0) \subseteq X_0$
  – So each iteration moves at least one step down the lattice (for some component of the vector)
  – $\ldots \subseteq F(F(X_0)) \subseteq F(X_0) \subseteq X_0$

• Therefore, # steps needed to reach a fixpoint is at most the height $H$ of $\mathcal{L}$ times the number of nodes: $O(Hn)$
QUALITY OF DATAFLOW ANALYSIS SOLUTIONS
Best Possible Solution

• Suppose we have a control-flow graph.
• If there is a path \( p_1 \) starting from the root node (entry point of the function) traversing the nodes \( n_0, n_1, n_2, \ldots, n_k \)
• The best possible information along the path \( p_1 \) is:
  \[ \ell_{p_1} = F_{n_k}(\ldots F_{n_2}(F_{n_1}(F_{n_0}(T)))\ldots) \]
• Best solution at the output is some \( \ell \sqsubseteq \ell_p \) for all paths \( p \).

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

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

• Does the iterative solution: \( \text{out}[n] = F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n']) \) compute the MOP solution?

• MOP Solution: \( \prod_{p \in \text{paths_to}[n]} \ell_p \)

• 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 control flow graph might mean there are infinitely many paths…)

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

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

- Does \( F_n \) distribute over meet \( \sqcap = \cup \)?

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

- Therefore: Reaching Definitions with iterative analysis always terminates with the MOP (i.e. best) solution.
“Classic” Constant Propagation

• Constant propagation can be formulated as a dataflow analysis.

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

• 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 this:

  \[
  \top = \text{multiple values} \\
  \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \\
  \bot = \text{never defined}
  \]

- 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 \( x = y \text{ op } z \)
• \( F(\ell_x, \ell_y, \ell_z) = ? \)

\[
\begin{align*}
F(\ell_x, T, \ell_z) &= (T, T, \ell_z) \\
F(\ell_x, \ell_y, T) &= (T, \ell_y, T)
\end{align*}
\]

“If either input might have multiple values the result of the operation might too.”

\[
\begin{align*}
F(\ell_x, \bot, \ell_z) &= (\bot, \bot, \ell_z) \\
F(\ell_x, \ell_y, \bot) &= (\bot, \ell_y, \bot)
\end{align*}
\]

“If either input is undefined the result of the operation is too.”

\[
F(\ell_x, i, j) = (i \text{ op } j, i, j)
\]

“If the inputs are known constants, calculate the output statically.”

• Flow functions for the other nodes are easy…
• Monotonic?
• Distributes over meets?
Iterative Solution

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

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

\[ (\bot, 1, 2) \sqcap (\bot, 2, 1) = (\bot, \top, \top) \]

\begin{align*}
&\text{iterative solution} \\
&\text{if } x > 0 \\
y & = 1 \\
z & = 2 \\
x & = y + z \\
\end{align*}

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

\[ (\bot, 1, 2) \sqcap (\bot, 2, 1) = (\bot, \top, \top) \]

\[ (\top, \top, \top) \]

\( (\top, \top, \top) \) iterative solution
MOP Solution ≠ Iterative Solution

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

MOP solution: \((3, 1, 2) \cap (3, 2, 1) = (3, T, T)\)
Why not compute MOP Solution?

- If MOP is better than the iterative analysis, why not compute it instead?
  - ANS: exponentially many paths (even in graph 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!
Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.
  - Solution is equivalent to meet-over-paths answer if the flow functions distribute over meet ($\sqcap$).

- Dataflow analyses as presented work for an “imperative” intermediate representation.
  - The 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 up: Converting to SSA, finding loops and dominators in CFGs