Another example: constant prop

- Set $D = \mathcal{P}(\{x \mapsto N \mid x \in \text{Var, } N \in \text{Int}\})$

\[
\begin{align*}
\text{F}_{X := N}(\text{in}) &= \text{in} - \{x \mapsto \star\} \\
&\quad \cup \{x \mapsto N\}
\end{align*}
\]

\[
\begin{align*}
\text{F}_{X := Y \text{ op } Z}(\text{in}) &= \text{in} - \{x \mapsto \star\} \\
&\quad \cup \{x \mapsto N, y \mapsto N_1, z \mapsto N_2, e \mapsto \text{in}\}
\end{align*}
\]
Another example: constant prop

- Set $D = 2 \{ x \rightarrow N \mid x \in \text{Vars} \land N \in \mathbb{Z} \}$

\[
X := N \\
\textbf{in} \quad \textbf{out}
\]

\[
F_{X := N}(\text{in}) = \text{in} - \{ X \rightarrow \ast \} \cup \{ X \rightarrow N \}
\]

\[
X := Y \text{ op } Z \\
\textbf{in} \quad \textbf{out}
\]

\[
F_{X := Y \text{ op } Z}(\text{in}) = \text{in} - \{ X \rightarrow \ast \} \cup \\
\{ X \rightarrow N \mid (Y \rightarrow N_1) \in \text{in} \land \\
(Z \rightarrow N_2) \in \text{in} \land \\
N = N_1 \text{ op } N_2 \}
\]
Another example: constant prop

\[ F_X := \star_Y \text{(in)} = \text{im} - \{ x \rightarrow \star \} \]

\[ U \{ x \rightarrow \mathbb{N} \mid \forall \nu \in \text{MayPF}(\gamma). \nu \rightarrow \mathbb{N} \in \text{in} \} \]

\[ F_{\star_X} := \gamma \text{(in)} = \text{im} - \{ \nu \rightarrow \mathbb{N} \mid \nu \in \text{MayPF}(\alpha) \}
\]

\[ U \{ z \rightarrow \mathbb{N} \mid \gamma \rightarrow \mathbb{N} \in \text{im}
\]

\[ z \rightarrow \mathbb{N} \in \text{im} \]
Another example: constant prop

\[
F_X := \ast_Y \text{ (in)} = \text{in} - \{ X \to \ast \} \\
\quad \cup \{ X \to N \mid \forall Z \in \text{may-point-to}(Y) \} \\
\quad \quad \cup \{ (Z \to N) \in \text{in} \}
\]

\[
F_{\ast X} := Y \text{ (in)} = \text{in} - \{ Z \to \ast \mid Z \in \text{may-point}(X) \} \\
\quad \cup \{ Z \to N \mid Z \in \text{must-point-to}(X) \land Y \to N \in \text{in} \} \\
\quad \quad \cup \{ Z \to N \mid (Y \to N) \in \text{in} \land (Z \to N) \in \text{in} \}
\]
Another example: constant prop

\[
\begin{align*}
*X & := *Y + *Z \\
F_*X := *Y + *Z & (\text{in}) = \text{in} - \{ \ldots \} \\
X & := G(\ldots) \\
F_X := G(\ldots) & (\text{in}) = \emptyset
\end{align*}
\]
Another example: constant prop

\[ *X := *Y + *Z \]

\[ F_X := G(...) \]

\[ a := *Y; b := *Z; c := a + b; *X := c \]

\[ F^*X := *Y + *Z \text{ (in)} = F \]

\[ F^a := *Y; b := *Z; c := a + b; *X := c \text{ (in)} \]

\[ F^X := G(...) \text{ (in)} = \emptyset \]
Another example: constant prop

\{ \gamma \rightarrow 0, \; x \rightarrow 10 \}\n
\text{in}

s: \text{if } (\ldots)

\text{out[0]}

\{ \gamma \rightarrow 0, \; x \rightarrow 10 \}\n
\text{out[1]}

\{ \gamma \rightarrow 0, \; x \rightarrow 10 \}\n
\text{merge}

\text{in[0]} \quad \text{in[1]}

\Rightarrow \wedge

\text{out}
Lattice

- \( (D, \sqsubseteq, \bot, T, [\sqcap], \sqcup) = \)

\[ \emptyset \]

\[ \emptyset \setminus \{ h < N \mid x \in \text{even} \} \]

FS

FS
Lattice

- \((D, \sqsubseteq, \bot, T, U, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)
  
  where \(A = \{ x \to \mathbb{N} | x \in \text{Vars} \land N \in \mathbb{Z} \}\)
Example

\[
f(a, b, c) = \begin{cases} 
  0 & \text{if } x = 5, v = 2 \\
  \{x \rightarrow 5\} & \text{if } x = 5, v \rightarrow 2 \\
  \{x \rightarrow 5, v \rightarrow 2\} & \text{otherwise}
\end{cases}
\]

1st

\[
x := x + 1 \\
w := v + 1
\]

FS

2nd

\[
x := 5 \\
v := 2
\]

\[
\{x \rightarrow 5\} \quad \{x \rightarrow 5, v \rightarrow 2\}
\]

\[
x := 6, v \rightarrow 2 \\
x := 5, v \rightarrow 2
\]

\[
\{x \rightarrow 6, v \rightarrow 2\}
\]

\[
\{x \rightarrow 5, v \rightarrow 2\}
\]

\[
\{w \rightarrow 3, y \rightarrow 10, z \rightarrow 15, x \rightarrow 5, v \rightarrow 2\}
\]

\[
\text{w := w * v}
\]

\[
w := 3 \\
y := x \times 2 \\
z := y + 5
\]
Another Example

\begin{align*}
x &:= 5 \\
a &:= x + 10 \\
x &:= x + 1 \\
x &:= x - 1 \\
b &:= x + 10
\end{align*}
Another Example starting at top

\[
\begin{align*}
x &:= 5 \\
a &:= x + 10
\end{align*}
\]

\[
\begin{align*}
x &:= x + 1 \\
x &:= x - 1
\end{align*}
\]

\[
b := x + 10
\]
Back to lattice

- \((D, \sqsubseteq, \bot, T, \cup, \cap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)
  where \(A = \{ x \to N \mid x \in \text{Vars} \land N \in Z \}\)

- What’s the problem with this lattice?
Back to lattice

- \((D, \sqsubseteq, \bot, T, U, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)
  where \(A = \{ x \rightarrow N \mid x \in \text{Vars} \land N \in \mathbb{Z} \}\)

- What’s the problem with this lattice?

- Lattice is infinitely high, which means we can’t guarantee termination
Better lattice

- Suppose we only had one variable
Better lattice

- Suppose we only had one variable

- $D = \{ \bot, \top \} \cup \mathbb{Z}$

- $\forall i \in \mathbb{Z} . \bot \leq i \land i \leq \top$

- height = 3
For all variables

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices \((D_1, \sqsubseteq_1, \bot_1, \top_1, \cup_1, \cap_1) \ldots (D_n, \sqsubseteq_n, \bot_n, \top_n, \cup_n, \cap_n)\) create:

  tuple lattice \(D^n = \)
For all variables

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices \((D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1) \ldots (D_n, \sqsubseteq_n, \bot_n, \top_n, \sqcup_n, \sqcap_n)\) create:

  tuple lattice \(D^n = ((D_1 \times \ldots \times D_n), \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) where
  \[
  \bot = (\bot_1, \ldots, \bot_n)
  \]
  \[
  \top = (\top_1, \ldots, \top_n)
  \]
  \[
  (a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup_1 b_1, \ldots, a_n \sqcup_n b_n)
  \]
  \[
  (a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap_1 b_1, \ldots, a_n \sqcap_n b_n)
  \]
  \[
  \text{height} = \text{height}(D_1) + \ldots + \text{height}(D_n)
  \]
For all variables

- Option 2: Map from variables to single lattice
- Given lattice \((D, \sqsubseteq_1, \bot_1, \top_1, \cup_1, \cap_1)\) and a set \(V\), create:

  \[
  \text{map lattice } V \rightarrow D = (V \rightarrow D, \sqsubseteq, \bot, \top, \cup, \cap)
  \]
Back to example

\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z \text{ (in)} = \text{in} \left[ x \text{ in } (Y) \overset{\text{op}}{\rightarrow} \text{in} (Z) \right] \]
Back to example

\[
\begin{align*}
X & := Y \circ Z \\
\xrightarrow{\text{in}} & in \\
\xrightarrow{\text{out}} & out
\end{align*}
\]

\[
F_X := Y \circ Z \text{(in)} = \text{in} \left[ X \rightarrow \text{in}(Y) \circ \text{in}(Z) \right]
\]

where \( a \circ b = \)

\[
\begin{array}{cccc}
\circ & d_1 & T \\
\circ & d_2 & T \\
d_1 & d_2 & T \circ d_1 & T \\
T & T & T & T
\end{array}
\]
General approach to domain design

• Simple lattices:
  – boolean logic lattice
  – powerset lattice
  – incomparable set: set of incomparable values, plus top and bottom (eg const prop lattice)
  – two point lattice: just top and bottom

• Use combinators to create more complicated lattices
  – tuple lattice constructor
  – map lattice constructor
May vs Must

• Has to do with definition of computed info

• Set of $x \rightarrow y$ must-point-to pairs
  – if we compute $x \rightarrow y$, then, then during program execution, $x$ must point to $y$

• Set of $x \rightarrow y$ may-point-to pairs
  – if during program execution, it is possible for $x$ to point to $y$, then we must compute $x \rightarrow y$
# May vs must

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>most optimistic (bottom)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>most conservative (top)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## May vs must

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>most optimistic</td>
<td>empty set</td>
<td>full set</td>
</tr>
<tr>
<td>(bottom)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>most conservative</td>
<td>full set</td>
<td>empty set</td>
</tr>
<tr>
<td>(top)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>overly big</td>
<td>overly small</td>
</tr>
<tr>
<td>merge</td>
<td>$\cup$</td>
<td>$\cap$</td>
</tr>
</tbody>
</table>
Common Sub-expression Elim

- Want to compute when an expression is available in a var
- Domain:
Common Sub-expression Elim

- Want to compute when an expression is available in a var
- Domain:

\[ S = \{ x \rightarrow E \mid x \in \text{Var}, E \in \text{Exp} \} \]

\[ 0 = 2 \]
\[ f = S \]
\[ T = \emptyset \]
\[ u = \land \]
Flow functions

\[
\begin{align*}
\text{X} & := \text{Y} \text{ op } Z \\
\text{F}_X & := \text{Y} \text{ op } Z \text{(in)} = \\
\text{F}_X & := \text{Y} \text{(in)} =
\end{align*}
\]
Flow functions

\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z(\text{in}) = \text{in} - \{ X \to * \} - \{ * \to ... X ... \} \cup \{ X \to Y \text{ op } Z \mid X \neq Y \land X \neq Z \} \]

\[ a := \theta + c \quad a \to \theta + c \]
\[ d := a + c \]

\[ \{ a \to \theta + c \}
\[ \{ d \to a + c \} \]

\[ X := Y \]

\[ F_X := Y(\text{in}) = \text{in} - \{ X \to * \} - \{ * \to ... X ... \} \cup \{ X \to E \mid Y \to E \in \text{in} \} \]
Example

\[
x := \text{read()}
\]
\[
v := a + b
\]
\[
x := x + 1
\]
\[
w := x + 1
\]
\[
a := w
\]
\[
v := a + b
\]
\[
z := x + 1
\]
\[
t := a + b
\]
\[
\emptyset
\]
Direction of analysis

• Although constraints are not directional, flow functions are.

• All flow functions we have seen so far are in the forward direction.

• In some cases, the constraints are of the form
  \[ \text{in} = F(\text{out}) \]

• These are called backward problems.

• Example: live variables
  – compute the set of variables that may be live.
Live Variables

- A variable is live at a program point if it will be used before being redefined.
- A variable is dead at a program point if it is redefined before being used.

\[
x \text{ live} \\
x \text{ dead} \\
x \text{ dead} \\
x = a + b \\
x = c + d
\]
Example: live variables

- Set \( D = \)
- Lattice: \( (D, \subseteq, \perp, T, \cup, \cap) = \)
Example: live variables

- Set $D = 2^\text{Vars}$
- Lattice: $(D, \subseteq, \bot, T, \cup, \cap) = (2^\text{Vars}, \subseteq, \emptyset, \text{Vars}, \cup, \cap)$

\[
\begin{align*}
X & := Y \text{ op } Z \\
F_X & := Y \text{ op } Z \text{(out)} =
\end{align*}
\]
Example: live variables

• Set $D = 2^{\text{Vars}}$

• Lattice: $(D, \subseteq, \bot, \top, \cup, \cap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, \cup, \cap)$

\[
\begin{array}{c}
\text{F}_{X} := Y \text{ op } Z(\text{out}) = \text{out} - \{X\} \cup \{Y, Z\}
\end{array}
\]
Example: live variables

\[
\begin{align*}
x &:= 5 \\
y &:= x + 2
\end{align*}
\]

\[
\begin{align*}
x &:= x + 1 \\
y &:= x + 10
\end{align*}
\]

\[
\ldots \quad y \quad \ldots
\]
Example: live variables

How can we remove the \( x := x + 1 \) stmt?
Revisiting assignment

\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z(\text{out}) = \text{out} - \{ X \} \cup \{ Y, Z\} \]
Revisiting assignment

\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z \text{(out)} = \text{out} - \{ X \} \cup \{ Y, Z \} \]

\[ \text{out} - \{ x \} \cup \]

\[ x \notin \text{out} \quad \varnothing : \{ Y, Z \} \]
Theory of backward analyses

- Can formalize backward analyses in two ways
- Option 1: reverse flow graph, and then run forward problem
- Option 2: re-develop the theory, but in the backward direction
Precision

• Going back to constant prop, in what cases would we lose precision?
Precision

• Going back to constant prop, in what cases would we lose precision?

```plaintext
x := 5
if (<expr>) {
    x := 6
}
... x ...

where <expr> is equiv to false
```

```plaintext
if (p) {
    x := 5;
} else {
    x := 4;
}
...

if (...) {
    x := -1;
} else {
    x := 1;
}

y := x * x;
... y ...
```
Precision

• The first problem: Unreachable code
  – solution: run unreachable code removal before
  – the unreachable code removal analysis will do its best, but may not remove all unreachable code

• The other two problems are path-sensitivity issues
  – Branch correlations: some paths are infeasible
  – Path merging: can lead to loss of precision
MOP: meet over all paths

- Information computed at a given point is the meet of the information computed by each path to the program point

```plaintext
if (...) {
    x := -1;
} else {
    x := 1;
}
y := x * x;
... y ...
```
MOP

• For a path $p$, which is a sequence of statements $[s_1, ..., s_n]$, define: $F_p(\text{in}) = F_{s_n}(...F_{s_1}(\text{in})...)$

• In other words: $F_p = F_{s_1} \circ ... \circ F_{s_n}$

• Given an edge $e$, let paths-to($e$) be the (possibly infinite) set of paths that lead to $e$

• Given an edge $e$, $\text{MOP}(e) = \bigcup \{ F_p(\bot) \mid p \in \text{paths-to}(e) \}$

• For us, should be called JOP (ie: join, not meet)
MOP vs. dataflow

• MOP is the “best” possible answer, given a fixed set of flow functions
  – This means that MOP ⊆ dataflow at edge in the CFG

• In general, MOP is not computable (because there can be infinitely many paths)
  – vs dataflow which is generally computable (if flow fns are monotonic and height of lattice is finite)

• And we saw in our example, in general, MOP ≠ dataflow
MOP vs. dataflow

• However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?

Dataflow

\[
x := -1; \\
x := 1; \\
\text{Merge} \\
y := x \times x; \\
\ldots \ y \ldots
\]

MOP

\[
x := -1; \\
x := 1; \\
y := x \times x; \\
\ldots \ y \ldots
\]
MOP vs. dataflow

- However, it would be great if by imposing some restrictions on the flow functions, we could guarantee that dataflow is the same as MOP. What would this restriction be?

- Distributive problems. A problem is distributive if:

  \[ \forall a, b . F(a \sqcup b) = F(a) \sqcup F(b) \]

- If flow function is distributive, then MOP = dataflow
Summary of precision

• Dataflow is the basic algorithm

• To basic dataflow, we can add path-separation
  – Get MOP, which is same as dataflow for distributive problems
  – Variety of research efforts to get closer to MOP for non-distributive problems

• To basic dataflow, we can add path-pruning
  – Get branch correlation

• To basic dataflow, can add both:
  – meet over all feasible paths