Another example: constant prop

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

\[
F_X := N(\text{in}) = \text{im} - \{x \mapsto *\} \cup \{x \mapsto N\}
\]

\[
F_X := Y \text{ op } Z(\text{in}) = \text{im} - \{x \mapsto *\} \cup \{x \mapsto "N_1 \text{ op } N_2"\} \cup \{y \mapsto N_1 \in \text{im} Y\} \cup \{z \mapsto N_2 \in \text{im} Z\}
\]
Another example: constant prop

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

\[
\begin{align*}
X &:= N \\
&
\text{in} \downarrow \quad \text{out} \\
F_{X := N} &:= N \text{(in)} = \text{in} - \{ X \rightarrow * \} \cup \{ X \rightarrow N \} \\
&
\text{in} \downarrow \quad \text{out} \\
X &:= Y \text{ op } Z \\
&
\text{in} \downarrow \quad \text{out} \\
F_{X := Y \text{ op } Z} &:= Y \text{ op } Z \text{(in)} = \left( \text{in} - \{ X \rightarrow * \} \right) \cup \{ X \rightarrow N \mid (Y \rightarrow N_1) \in \text{in} \land (Z \rightarrow N_2) \in \text{in} \land N = N_1 \text{ op } N_2 \} \\
\end{align*}
\]

- $C := a + b$

- $\{a \rightarrow 5, b \rightarrow 10\}$

- $\{a \rightarrow 15, b \rightarrow 10\}$

- $\{b \rightarrow 10\}$

- $\{a \rightarrow 15\}$
Another example: constant prop

\[ F_X := *_Y \text{in} = \text{im} - \{ X \rightarrow * \} \cup \{ X \rightarrow N \mid \forall \; \nu \in \text{may-\text{pt}}(Y) \; \nu \rightarrow N \in \text{im} \} \]

\[ F_*X := Y \text{in} = \text{im} - \{ Z \rightarrow * \mid Z \in \text{mpt}(x) \} \cup \]

\[ Z \rightarrow N \]

\[ X \rightarrow N \]
Another example: constant prop

\[
\begin{align*}
F_{X := \ast Y} \text{ (in)} &= \text{in} - \{ X \rightarrow \ast \} \\
&\quad \cup \{ X \rightarrow N \mid \forall Z \in \text{may-point-to}(Y) . \ (Z \rightarrow N) \in \text{in} \} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{in} & \quad \rightarrow \quad \text{out} \\
\text{in} & \quad \rightarrow \quad \text{out}
\end{align*}
\]

\[
\begin{align*}
*X := & \quad *Y + *Z \\
F_{*X} := & \quad *Y + *Z(\text{in}) =
\end{align*}
\]

\[
\begin{align*}
\text{in} & \quad \rightarrow \quad \text{out} \\
\text{in} & \quad \rightarrow \quad \text{out}
\end{align*}
\]

\[
\begin{align*}
X := & \quad G(\ldots) \\
F_{X} := & \quad G(\ldots)(\text{in}) =
\end{align*}
\]
Another example: constant prop

\[
\begin{align*}
X := G(\ldots) & \quad F_X := G(\ldots)(\text{in}) = \emptyset \\
\text{in} & \quad \text{out} \\
*X := *Y + *Z & \quad F_{*X} := *Y + *Z(\text{in}) = F_{a := *Y; b := *Z; c := a + b; *X := c}(\text{in}) \\
\text{in} & \quad \text{out} \\
\end{align*}
\]
Another example: constant prop
Lattice

\[(D, \sqsubseteq, \perp, T, U, \sqcap, \sqcup) = \]

\[\emptyset, \geq, AS, \emptyset, \land, \lor\]

\[\emptyset\]

\[\{ x \in \mathbb{N} \mid x \in \text{val}, N \in \text{Int} \} \]
Lattice

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

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

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

\[ w := 3 \]
\[ w := w \times v \]

\[ y := x \times 2 \]
\[ z := y + 5 \]

\[ u \cup V = V \]
Another Example

```plaintext
x := 5  \(\xrightarrow{5}\)  \{x \rightarrow 5\} \{a \rightarrow 15\}
a := x + 10

x := x + 1  \(\xrightarrow{6}\)  \{x \rightarrow 6, a \rightarrow 15\}
x := x - 1  \(\xrightarrow{5}\)  \{x \rightarrow 5, a \rightarrow 15\}

b := x + 10  \(\xrightarrow{5, a \rightarrow 15}\)
```
x := 5
a := x + 10
x := x + 1
x := x - 1
2 := 10

Another Example starting at top
Back to lattice

• \((D, \sqsubseteq, \bot, \top, \cup, \cap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)
  where \(A = \{ x \rightarrow \mathbb{N} | x \in \text{Vars} \land N \in \mathbb{Z} \}\)

• What’s the problem with this lattice?
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

\[
\emptyset
\]

\[
\{4\} \quad \{5\} \quad \{6\} \quad \{7\} \quad \{8\}
\]

\[
\{5, 6\}
\]

\[
\{5, 6, 7\}
\]
Better lattice

- Suppose we only had one variable

\[ D = \{ \bot, T \} \cup \mathbb{Z} \]

- \( \forall i \in \mathbb{Z} . \bot \leq i \wedge i \leq T \)

- 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 \((D_1, \sqsubseteq_1, \bot_1, \top_1, \cup_1, \cap_1)\) ... \((D_n, \sqsubseteq_n, \bot_n, \top_n, \cup_n, \cap_n)\) create:

  tuple lattice \(D^n = ((D_1 \times \ldots \times D_n), \sqsubseteq, \bot, \top, \cup, \cap)\) where

  \(\bot = (\bot_1, \ldots, \bot_n)\)

  \(\top = (\top_1, \ldots, \top_n)\)

  \((a_1, ..., a_n) \cup (b_1, ..., b_n) = (a_1 \cup_1 b_1, ..., a_n \cup_n b_n)\)

  \((a_1, ..., a_n) \cap (b_1, ..., b_n) = (a_1 \cap_1 b_1, ..., a_n \cap_n b_n)\)

  height = height(D_1) + ... + height(D_n)
For all variables

- Option 2: Map from variables to single lattice
- Given lattice \((D, \subseteq_1, \bot_1, \top_1, \cup_1, \cap_1)\) and a set \(V\), create:

\[
\text{map lattice } V \rightarrow D = (V \rightarrow D, \subseteq, \bot, \top, \cup, \cap)
\]
\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z(\text{in}) = \text{im} \left[ X \mapsto \text{in}(\gamma) \hat{\phi} \text{in}(\lambda) \right] \]

\[
\begin{array}{c|ccc}
\text{op} & \bot & b & T \\
\hline
\bot & \bot & T & T \\
b & T & a \text{ op } b & T \\
T & T & T & T \\
\end{array}
\]

\[ 0 \ast \bot \]
Back to example

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

where \( \widehat{\circ} \) is defined by the following truth table:

<table>
<thead>
<tr>
<th>( \circ )</th>
<th>( d_1 )</th>
<th>( d_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \circ )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( d_2 )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
</tbody>
</table>
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

\[ \{ a \rightarrow b, a \rightarrow c \} \]

• 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><strong>most optimistic</strong></td>
<td>ᵃ</td>
<td>FS</td>
</tr>
<tr>
<td><em>(bottom)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>most conservative</strong></td>
<td>x ↦ y, y ↦ x ...</td>
<td>Ø</td>
</tr>
<tr>
<td><em>(top)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>safe</td>
<td>add</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{array}{c}
\{y \mapsto 10\} \\
* x := 5 \\
\text{print}(y)
\end{array} \]

\( \{x \mapsto a, x \mapsto e\} \)
<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>most optimistic (bottom)</td>
<td>empty set</td>
<td>full set</td>
</tr>
<tr>
<td>most conservative (top)</td>
<td>full set</td>
<td>empty set</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:

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

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

\[
F_X := Y \text{ op } Z \text{(in)} = \{ \text{in} \}
\]

\[
F_X := Y \text{ (in)} = \{ \text{in} \} - \{ \{ X \rightarrow \ast \} \}
\]

\[
a := b + c \quad \{ a = b + c \}
\]

\[
b := e + f
\]

\[
-\{ a \rightarrow b + c, b \rightarrow e + f \}
\]

\[
l + c \rightarrow a
\]

\[
\bigcup \{ X \rightarrow Y \text{ op } Z \} \\
\land Y \notin X \land Z \notin X
\]

\[
im - \{ \{ X \rightarrow \ast \} \} \\
\bigcup \{ X \rightarrow E \mid Y \rightarrow E \\
E \in \text{im} \}
\]
Flow functions

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

\[ F_X := Y(\text{in}) = \text{in} - \{ X \rightarrow * \} - \{ * \rightarrow \ldots X \ldots \} \cup \{ X \rightarrow E \mid Y \rightarrow E \in \text{in} \} \]

\[ c \rightarrow a \cdot f \quad b \rightarrow a + b \]

\[ a := c + b \]

\[ a \rightarrow (a \cdot f) + (a \cdot b) \]
Example

\[
\begin{align*}
x &:= \text{read()} \\
v &:= a + b \\
x &:= x + 1 \\
w &:= x + 1 \\
\end{align*}
\]

\[
\begin{align*}
w &:= x + 1 \\
a &:= w \\
v &:= a + b \\
z &:= x + 1 \\
t &:= a + b \\
\end{align*}
\]
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.

```
int x;

x := 5

x := 1

print(x)

x := 2

print(x)

x := 1

x := 5

print(x)    // Live
```
Example: live variables

- Set $D =$
- Lattice: $(D, \subseteq, \bot, \top, \cup, \cap) = \mathcal{P}(\nu m), \cup, \cap$
Example: live variables

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

\[
\begin{align*}
X & \leftarrow Y \oplus Z \\
\text{in} & \quad \text{out} \\
\alpha & \leftarrow b + c \\
F_X := Y \oplus Z(\text{out}) & = \text{out} - \{x\} \\
& \quad \cup \{y, z\}
\end{align*}
\]
Example: live variables

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

\[
\begin{array}{c}
\text{in} \\
X := Y \text{ op } Z \\
\text{out}
\end{array}
\]

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

\[ x := 5 \]
\[ y := x + 2 \]
\[ z := 0 \]
\[ x := x + 1 \]
\[ y := x + 10 \]

\[ \ldots y \ldots \]
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

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

\[ \rightarrow b := \ldots \leftarrow \]

\[ a := \{ d, e \} \]

\[ a := a + c \leftarrow \{ d, e \} \]

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

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

\[ x \notin \text{out} \ ? \emptyset : \{ Y, Z \} \]
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

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

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: \( F_p = \bigcirc_{s_1} \circ ... \circ F_{s_n} \)

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

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

• 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 $\subseteq$ 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 $\neq$ 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

\[
\begin{align*}
x &:= -1; \\
x &:= 1; \\
&\text{Merge} \\
y &:= x \times x; \\
&\text{... y ...}
\end{align*}
\]

MOP

\[
\begin{align*}
x &:= -1; \\
x &:= 1; \\
y &:= x \times x; \\
&\text{... y ...}
\end{align*}
\]

\[
\begin{align*}
&\text{Merge}
\end{align*}
\]
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