Another example: constant prop

\[ \text{Set } D = \{ x \rightarrow N | x \in \text{Vars} \land N \in \mathbb{Z} \} \]

\[
\begin{align*}
\text{in} & \quad \text{out} \\
X := N & \quad F_X := N \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ x 
\end{align*}
\]

\[
\begin{align*}
\text{in} & \quad \text{out} \\
X := Y \text{ op } Z & \quad F_X := Y \text{ op } Z \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ Y := N, \text{ in } Z \rightarrow N, \text{ in } \}
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := *Y & \quad F_X := *Y \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ X := N, \text{ in } Y := N \land \}
\end{align*}
\]

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := Y & \quad F_X := Y \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ Z := N \land \}
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := *Y + *Z & \quad F_X := *Y + *Z \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ X := N, \text{ in } Y := N \land \}
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := *Y + *Z & \quad F_X := *Y + *Z \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ X := N, \text{ in } \}
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
X := G(\ldots) & \quad F_X := G(\ldots) \text{ (in)} = \emptyset
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
X := G(\ldots) & \quad F_X := G(\ldots) \text{ (in)} = \emptyset
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := *Y + *Z & \quad F_X := *Y + *Z \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ X := N, \text{ in } Y := N \land \}
\end{align*}
\]

Another example: constant prop

\[
\begin{align*}
\text{in} & \quad \text{out} \\
*X := *Y + *Z & \quad F_X := *Y + *Z \text{ (in)} = \text{id} \cap \{ x \rightarrow N \} \cup \{ X := N, \text{ in } \}
\end{align*}
\]
Another example: constant prop

```
\begin{align*}
\{ \gamma \rightarrow 0, \kappa \rightarrow \text{true} \} \\
\downarrow \text{if (\text{true})} \\
out[0] \\
\{ \gamma \rightarrow 0, \kappa \rightarrow \text{false} \} \\
\downarrow \text{merge} \\
\downarrow \text{out}
\end{align*}
```

Lattice

```
\text{Lattice} = \text{Lattice}(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) = (\mathbb{A}, \text{\textyen}, \mathbb{A}, \text{\textvar}, \text{\textvar})
```

where $\mathbb{A} = \{ x \rightarrow \mathbb{N} \mid x \in \text{Vars} \land \mathbb{N} \in \mathbb{Z} \}$

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 \\
x &:= x + 1 \\
x &:= x - 1 \\
b &:= x + 10
\end{align*}
```
• (D, ⊑, ⊥, ⊤, ⊔, ⊓) = (2^A, ⊇, A, ∅, ⊔, ⊓)
  where \( A = \{ x \rightarrow N | x \in \text{Vars} \land N \in \mathbb{Z} \} \)

• What’s the problem with this lattice?

• Lattice is infinitely high, which means we can’t guarantee termination

• Suppose we only had one variable

• D = {⊥, ⊤} \( \cup \mathbb{Z} \)
  \( \forall i \in \mathbb{Z} \colon \perp \leq i \leq \top \)
  height = 3

• Two possibilities
  Option 1: Tuple of lattices
  Given lattices (\( D_1, \bot_1, \top_1, \land_1, \lor_1, \lnot_1 \)) ...
  (\( D_n, \bot_n, \top_n, \land_n, \lor_n, \lnot_n \)) create:

  Tuple lattice \( D^n = \)

• For all variables

• Two possibilities
  Option 1: Tuple of lattices
  Given lattices (\( D_1, \bot_1, \top_1, \land_1, \lor_1, \lnot_1 \)) ...
  (\( D_n, \bot_n, \top_n, \land_n, \lor_n, \lnot_n \)) create:

  Tuple lattice \( D^n = (D_1 \times \ldots \times D_n, \bot, \top, \land, \lor, \lnot) \) where

  \[ \bot = (\bot_1, \ldots, \bot_n) \]
  \[ \top = (\top_1, \ldots, \top_n) \]
  \[ (a_1, \ldots, a_n) \land (b_1, \ldots, b_n) = (a_1 \land b_1, \ldots, a_n \land b_n) \]
  \[ (a_1, \ldots, a_n) \lor (b_1, \ldots, b_n) = (a_1 \lor b_1, \ldots, a_n \lor b_n) \]
  \[ \text{height} = \text{height}(D_1) + \ldots + \text{height}(D_n) \]
For all variables

- Option 2: Map from variables to single lattice
- Given lattice \((D, \subseteq_1, \bot_1, T_1, \sqcup_1, \sqcap_1)\) and a set \(V\), create:

\[
\text{map lattice } V \rightarrow D = (V \rightarrow D, \subseteq, \bot, T, \sqcup, \sqcap)
\]

General approach to domain design

- Simple lattices:
  - boolean logic lattice
  - powerset lattice
  - incomparable set: set of incomparable values, plus top and bottom (e.g., 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 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 \(x \rightarrow y\)

<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 (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:

\[
\mathcal{S} = \left\{ X \in \mathcal{E} \mid X \in \text{let }, E \in \mathcal{E} \right\}
\]

\[
\begin{align*}
\emptyset & \subseteq \mathcal{S} \\
\{ S \} & \subseteq \mathcal{S} \\
\mathcal{S} & \subseteq \mathcal{S} \\
\emptyset & \not\subseteq \mathcal{S}
\end{align*}
\]

Flow functions

\[
\begin{align*}
F_{X := Y \text{ op } Z} \text{(in)} &= \text{in} - \left\{ X \rightarrow ^{*} \right\} \\
&\quad - \left\{ ^{*} \rightarrow \ldots X \ldots \right\} \cup \\
&\quad \left\{ X \rightarrow \text{ op } Z \mid X \neq Y \land X \neq Z \right\}
\end{align*}
\]

\[
\begin{align*}
F_{X := Y} \text{(in)} &= \text{in} - \left\{ X \rightarrow ^{*} \right\} \\
&\quad - \left\{ ^{*} \rightarrow \ldots X \ldots \right\} \cup \\
&\quad \left\{ X \rightarrow E \mid Y \rightarrow E \in \text{in} \right\}
\end{align*}
\]

Example

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

Example: live variables

- Set \( D = \)
- Lattice: \((D, \subseteq, \top, T, U, \cap) = \)

Example: live variables

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

Example: live variables

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

Example: live variables

- Set \( D = 2^{\text{Vars}} \)
- Lattice: \((D, \subseteq, \top, T, U, \cap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, U, \cap) \)
Example: live variables

Revisiting assignment

Revisiting assignment

Theory of backward analyses

Precision

Precision
### 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 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?

### 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