Another example: constant prop

\[ D = \{ x \rightarrow C \mid x \in \text{Vars}, C \in \text{Cont} \} \]

\[ X := N \]

\[ F_{X \rightarrow \text{in}(\text{in})} = \{ \text{in} \} \]

\[ X := Y \oplus Z \]

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

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

\[ F_{X \rightarrow \text{in}(\text{in})} = \emptyset \]

Another example: constant prop

\[ D = 2 \{ x \rightarrow N \mid x \in \text{Vars}, N \in \mathbb{Z} \} \]

\[ X := N \]

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

\[ X := Y \oplus Z \]

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

\[ X := *Y \]

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

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

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

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

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

Another example: constant prop

\[ D = \{ x \rightarrow 2 \mid x \in \text{Vars}, x \in \mathbb{N} \} \]

\[ X := N \]

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

\[ X := Y \oplus Z \]

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

\[ X := *Y \]

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

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

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

\[ *X := *Y \]

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

\[ *X := N \]

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

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

\[ F_{X \rightarrow \text{in}(\text{in})} = \emptyset \]

Another example: constant prop

\[ D = \{ x \rightarrow 0 \mid x \in \text{Vars}, x \in \mathbb{N} \} \]

\[ X := N \]

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

\[ X := Y \oplus Z \]

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

\[ X := *Y \]

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

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

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

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

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

Another example: constant prop

\[ D = \{ x \rightarrow 1 \mid x \in \text{Vars}, x \in \mathbb{N} \} \]

\[ X := N \]

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

\[ X := Y \oplus Z \]

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

\[ X := *Y \]

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

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

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

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

\[ F_{X \rightarrow \text{in}(\text{in})} = \{ \text{in} \} \cup \{ X \rightarrow N \} \]
Another example: constant prop

Lattice

• \((D, \sqsubseteq, \bot, T, U, \Pi) =\)

Example

Another Example starting at top
• \((D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\) 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

Better lattice

• Suppose we only had one variable

- \(D = \{ \bot, \top \} \cup \mathbb{Z} \)
- \(\forall i \in \mathbb{Z} : \bot \sqsubseteq i \sqsubseteq \top \)
- height = 3

For all variables

• Two possibilities
• Option 1: Tuple of lattices
• Given lattices \((D_1, \sqsubseteq, \bot, \top, \sqcup, \sqcap) \ldots (D_n, \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) where \(\bot = (\bot_1, \ldots, \bot_n)\)
- \(T = (T_1, \ldots, T_n)\)
- \((a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \cup b_1, \ldots, a_n \cup b_n)\)
- \((a_1, \ldots, a_n) \cap (b_1, \ldots, b_n) = (a_1 \cap b_1, \ldots, a_n \cap b_n)\)
- height = \(\text{height}(D_1) + \ldots + \text{height}(D_n)\)
For all variables

- Option 2: Map from variables to single lattice
- Given lattice \((D, \subseteq, \bot, T, \cup, \cap, \top)\) and a set \(V\), create:

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

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 \(x \rightarrow y\)

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>most optimistic (bottom)</td>
<td>(\emptyset)</td>
<td>FS</td>
</tr>
<tr>
<td>most conservative (top)</td>
<td>FS</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>safe</td>
<td>Add</td>
<td>Unsafe</td>
</tr>
<tr>
<td>merge</td>
<td>U</td>
<td>(\land)</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:

\[
S = \{ X = E \mid X \in \mathcal{L}_r, E \in \mathcal{E}_{r} \}
\]

\[
\emptyset \subseteq S(s)
\]

Flow functions

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

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

Example
**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 $in = F(out)$.
- 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 $D = \mathcal{P}(\mathcal{V})$
- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (\mathcal{P}(\mathcal{V}), \subseteq, \emptyset, \top)$

**Example: live variables**

- Set $D = 2^{\mathcal{V}}$
- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Set $D = 2^{\mathcal{V}}$
- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Set $D = 2^{\mathcal{V}}$
- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (\mathcal{P}(\mathcal{V}), \subseteq, \emptyset, \top)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (\mathcal{P}(\mathcal{V}), \subseteq, \emptyset, \top)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (\mathcal{P}(\mathcal{V}), \subseteq, \emptyset, \top)$

**Example: live variables**

- Lattice: $(D, \subseteq, \bot, \top, \setminus, \cap) = (2^{\mathcal{V}}, \subseteq, \emptyset, \setminus, \cap)$
Example: live variables

Revisiting assignment

Theory of backward analyses

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;
}
```

```
\begin{align*}
  x &:= -1; \\
  y &:= x \times x; \\
  ... y ... \\
  &= (x \times x);
\end{align*}
```

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?

  - Distributive problems. A problem is distributive if:
    \[ \forall a, b : F(a \cup b) = F(a) \cup F(b) \]

  - 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.