### Another example: constant prop

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

<table>
<thead>
<tr>
<th>$\text{in}$</th>
<th>$\text{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := N$</td>
<td>$F_{X := N}(\text{in}) = \text{in} - { x \rightarrow * } \cup { X \rightarrow N }$</td>
</tr>
<tr>
<td>$x := \gamma \rightarrow z$</td>
<td>$F_{X := \gamma \rightarrow z}(\text{in}) = \text{in} - { Z \rightarrow * } \cup { Z \rightarrow \gamma \rightarrow z }$</td>
</tr>
<tr>
<td>$\ast x := y$</td>
<td>$F_{\ast X := y}(\text{in})$</td>
</tr>
<tr>
<td>$\ast x := y + \ast z$</td>
<td>$F_{\ast X := y + \ast z}(\text{in})$</td>
</tr>
<tr>
<td>$x := G(\ldots)$</td>
<td>$F_{X := \mathcal{G}(\ldots)}(\text{in}) = \emptyset$</td>
</tr>
</tbody>
</table>
Another example: constant prop

Lattice
• \((D, \subseteq, \bot, T, U, \Pi) =\)

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

Example

Another Example

Another Example starting at top
Back to lattice

• \((D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) = (\mathcal{P}(\mathbb{N}), \subseteq, \emptyset, \mathbb{N}, \cup)\)
  where \(A = \{ x \rightarrow \mathbb{N} \mid x \in \text{Vars} \land N \in \mathbb{Z} \}\)

• What's the problem with this lattice?

Back to lattice

• \((D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) = (\mathcal{P}(\mathbb{N}), \subseteq, \emptyset, \mathbb{N}, \cup)\)
  where \(A = \{ x \rightarrow \mathbb{N} \mid x \in \text{Vars} \land N \in \mathbb{Z} \}\)

• 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

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 \(D^n = \)

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

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

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

Back to example

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

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 (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:

$$S = \{ X \Rightarrow E \mid X \in \text{vars}, E \in \text{expr} \}$$

Flow functions

$$F_{x \Rightarrow y \Rightarrow z}^{\text{in}} = \text{in} - \{ X \Rightarrow \}$

$$\quad - \{ X \Rightarrow Y \Rightarrow Z \} \cup \{ X \Rightarrow Y \Rightarrow Z \mid X \not= Y \land X \not= Z \}$$

$$F_{x \Rightarrow y}^{\text{in}} = \text{in} - \{ X \Rightarrow \}$

$$\quad - \{ X \Rightarrow Y \Rightarrow \} \cup \{ X \Rightarrow E \mid Y \Rightarrow E \text{ in } \text{in} \}$$

Example

```
x := read()
v := a + b
w := x + 1
x := x + 1
y := a + b
w := x + 1
a := w
t := x + 1
```
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 $D = 2^{Vars}$
- Lattice: $(D, \subseteq, \perp, T, U, \cap) = \{x := 5, y := x + 2\}$
- Lattice: $(D, \subseteq, \perp, T, U, \cap) = (2^{Vars}, \subseteq, \emptyset, Vars, U, \cap)$
- $X := Y \text{ op } Z$ in $out$
- $F_{X :\text{ op } Z}(out) = out - \{X\} \cup \{Y, Z\}$

Example: live variables

- Set $D = 2^{Vars}$
- Lattice: $(D, \subseteq, \perp, T, U, \cap) = (2^{Vars}, \subseteq, \emptyset, Vars, U, \cap)$
- $X := Y \text{ op } Z$ in $out$
- $F_{X :\text{ op } Z}(out) = out - \{X\} \cup \{Y, Z\}$

Example: 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

\[
\begin{align*}
  x &:= 5 \\
  y &:= x + 2 \\
  x &:= x + 1 \\
  y &:= x + 10 \\
\end{align*}
\]

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

Revisiting assignment

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

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?

\[
\begin{align*}
  x &:= 5 \\
  \text{if (expr)} \{} & x := 6 \\
  \text{else} & x := 4; \\
\end{align*}
\]

\[
\begin{align*}
  \text{if (p)} \{} & y := x + 1 \\
  \text{else} & y := x + 2; \\
\end{align*}
\]
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
  \([s_1, \ldots, s_n]\), define: \(F_p(in) = F_{s_n}(\ldots F_{s_1}(in) \ldots)\)
- In other words: \(F_p = F_{s_n} \circ \cdots \circ F_{s_1}\)
- Given an edge e, let paths-to(e) be the (possibly infinite) set of paths that lead to e
- Given an edge e, \(\text{MOP}(e) = \bigvee_{p \in \text{paths-to}(e)} F_p(\perp)\)
- 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 \(\text{MOP} \subseteq \text{dataflow} \) 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, \(\text{MOP} \neq \text{dataflow}\)

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?

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

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:

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

- If flow function is distributive, then \(\text{MOP} = \text{dataflow}\)
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