Another example: constant prop

Lattice

\[ (D, \sqsubseteq, \bot, T, U, \Pi) = \]

Lattice

\[ (D, \sqsubseteq, \bot, T, U, \Pi) = \]
\[ (2^A, \subseteq, A, \emptyset, \cap, \cup) \]

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

Example

Another Example

Another Example starting at top
Back to lattice

• \((D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) = (2^A, \sqsubseteq, \emptyset, \emptyset, \emptyset, \emptyset)\)
  where \(A = \{x \rightarrow N \mid x \in \text{Vars} \land N \in \mathbb{Z}\}\)

• What’s the problem with this lattice?

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 \((D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1) \ldots (D_m, \sqsubseteq_m, \bot_m, \top_m, \sqcup_m, \sqcap_m)\) create:
    tuple lattice \(D^n = ((D_1 \times \ldots \times D_m), \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) where
    \(\bot = (\bot_1, \ldots, \bot_m)\)
    \(\top = (\top_1, \ldots, \top_m)\)
    \((a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup b_1, \ldots, a_n \sqcup b_n)\)
    \((a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap b_1, \ldots, a_n \sqcap b_n)\)
    height = \(\text{height}(D_1) + \ldots + \text{height}(D_m)\)

For all variables

• Two possibilities
  • Option 1: Tuple of lattices
  • Given lattices \((D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1) \ldots (D_m, \sqsubseteq_m, \bot_m, \top_m, \sqcup_m, \sqcap_m)\) create:
    tuple lattice \(D^n = ((D_1 \times \ldots \times D_m), \sqsubseteq, \bot, \top, \sqcup, \sqcap)\) where
    \(\bot = (\bot_1, \ldots, \bot_m)\)
    \(\top = (\top_1, \ldots, \top_m)\)
    \((a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup b_1, \ldots, a_n \sqcup b_n)\)
    \((a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap b_1, \ldots, a_n \sqcap b_n)\)
    height = \(\text{height}(D_1) + \ldots + \text{height}(D_m)\)
For all variables

- Option 2: Map from variables to single lattice
- Given lattice $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \sqsupseteq, \sqsubseteq)$ and a set $V$, create:

$$\text{map lattice } V \rightarrow D \Rightarrow (V \rightarrow D, \sqsubseteq, \top, \sqcup, \sqcap, \sqsupseteq)$$

$$\bot = \lambda v \mapsto \bot,$$

Back to example

Back to example

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, 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>May vs must</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
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:

\[
S := \{ X = E | X \in \mathcal{L}_X, E \in E_{expr} \}
\]

\[
\begin{align*}
S & := \{ X = E | X \in \mathcal{L}_X, E \in E_{expr} \} \\
S & := \{ X = E | X \in \mathcal{L}_X, E \in E_{expr} \}
\end{align*}
\]

Flow functions

- \( F_{X := Y \circ Z}(in) = \)
- \( F_{X := Y}(in) = \)

Example

\[
\begin{align*}
x & := \text{read()} \\
v & := a + b \\
w & := x + 1 \\
a & := w \\
y & := a + b \\
t & := a + b
\end{align*}
\]
**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 $D = 2$.
- Lattice: $(D, \subseteq, \bot, T, U, \cap) = (2^{\text{Vars}}, \subseteq, \emptyset, \text{Vars}, U, \cap)$

---

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

- $X := Y \oplus Z$

- $F_{X \Rightarrow Y \oplus Z}(\text{out}) = \text{out} - \{X\} \cup \{Y, Z\}$

---

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

- $x := 5$
- $y := x + 2$
- $x := x + 1$
- $y := x + 10$
- $\ldots$ $y \ldots$

---

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

- $F_{X \Rightarrow Y \oplus Z}(\text{out}) = \text{out} - \{X\} \cup \{Y, Z\}$

---
**Example: live variables**

- $x := 5$
  - $y := x + 2$
  - $x := x + 1$
- $y := x + 10$

How can we remove the $x := x + 1$ stmt?

**Revisiting assignment**

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

**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?

- $x := 5$
  - if (expr) {
    - $x := 6$
    - ...
  } else {
    - ...
  }
  - where $\text{expr}$ is equiv to false

- if (p) {
  - $x := 5$
  - if (...) {
    - $x := -1$
    - } else {
      - $x := 1$
      - }
    - $y := x \times x$
  } else {
    - ...
  }

- if (p) {
  - $y := x + 1$
  - } else {
    - $y := x + 2$
    - }
  - ...

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: \(F_p(in) = F_{s_n}(...F_{s_1}(in)...)\)
- In other words: \(F_p = \bigcirc_{s_n} \circ \cdots \circ \bigcirc_{s_1}\)
- Given an edge e, let paths-to(e) be the (possibly infinite) set of paths that lead to e
- Given an edge e, \(MOP(e) = \bigoplus_{p \in \text{paths-to}(e)} F_p(\bot)\)
- 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 fn 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?

```
x := -1;
MOP
y := x * x;
... y ...
```

```
x := -1;
Dataflow  x := 1;
y := x * x;
... y ...
```

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