Another example: constant prop

- Set $D =$

\[
\begin{align*}
\text{in} \quad X &:= N \\
\text{out} \\
\text{in} \quad X &:= Y \text{ op } Z \\
\text{out} \\
\end{align*}
\]

$F_X := N(\text{in}) =$

$F_X := Y \text{ op } Z(\text{in}) =$
Another example: constant prop

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

\[
F_X := N \text{(in)} = \text{in} - \{ X \to * \} \cup \{ X \to N \}
\]

\[
F_X := Y \, \text{op} \, Z \text{(in)} = \text{in} - \{ X \to * \} \cup \\
\{ X \to N \mid (Y \to N_1) \in \text{in} \land \\
(Z \to N_2) \in \text{in} \land \\
N = N_1 \, \text{op} \, N_2 \}
\]
Another example: constant prop

\[ X := \ast Y \]

\[ F_X := \ast_Y (\text{in}) = \]

\[ \ast X := Y \]

\[ F_{\ast X} := Y (\text{in}) = \]
Another example: constant prop

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

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

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

\[ F^*_X := *Y + *Z \text{(in)} = \]

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

\[ F_X := G(...) \text{(in)} = \]
Another example: constant prop

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

\[F_{*X} := *Y + *Z \text{(in)} = F_{a := *Y; b := *Z; c := a + b; *X := c \text{ (in)}}
\]

\[F_X := G(\ldots) \text{(in)} = \emptyset\]
Another example: constant prop

\[ s: \text{if} (...) \]

\[ \text{out}[0] \quad \text{out}[1] \]

\[ \text{in} \]

\[ \text{merge} \]

\[ \text{in}[0] \quad \text{in}[1] \]

\[ \text{out} \]
Lattice

- $(D, \sqsubseteq, \bot, T, \sqcup, \sqcap)$ =
Lattice

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

\[
\begin{align*}
x &= 5 \\
v &= 2 \\
x &= x + 1 \\
w &= v + 1 \\
w &= 3 \\
y &= x \times 2 \\
z &= y + 5 \\
w &= w \times v
\end{align*}
\]
Another Example

\[
x := 5 \\
a := x + 10
\]

\[
x := x + 1 \\
x := x - 1
\]

\[
b := x + 10
\]
Another Example starting at top

\begin{align*}
x := 5 \\
a := x + 10 \\
x := x + 1 \\
x := x - 1 \\
b := x + 10
\end{align*}
Back to lattice

• \((D, \sqsubseteq, \bot, T, U, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)

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

• What’s the problem with this lattice?
Back to lattice

- \((D, \sqsubseteq, \perp, T, U, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)\)

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

- 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

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

- \( \forall i \in \mathbb{Z} . \bot \leq i \land i \leq \top \)

- height = 3
For all variables

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices \((D_1, \sqsubseteq_1, \bot_1, \top_1, \cup_1, \cap_1)\) ... \((D_n, \sqsubseteq_n, \bot_n, \top_n, \cup_n, \cap_n)\) create:

  tuple lattice \(D^n = \)
For all variables

- Two possibilities
- Option 1: Tuple of lattices
- Given lattices \((D_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1) \ldots (D_n, \sqsubseteq_n, \bot_n, \top_n, \sqcup_n, \sqcap_n)\) create:

\[
\text{tuple lattice } D^n = ((D_1 \times \ldots \times D_n), \sqsubseteq, \bot, \top, \sqcup, \sqcap) \text{ where }
\]

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

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

\[
(a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup_1 b_1, \ldots, a_n \sqcup_n b_n)
\]

\[
(a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap_1 b_1, \ldots, a_n \sqcap_n b_n)
\]

height = height(D_1) + \ldots + 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:

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

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

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

\[ F_X := Y \mathop{\text{op}} Z \text{(in)} = \text{in} \left[ X \rightarrow \text{in}(Y) \mathop{\text{op}} \text{in}(Z) \right] \]

where \( a \mathop{\text{op}} b = \)

\[
\begin{array}{c c c c}
\hat{\mathop{\text{op}}} & 1 & d_1 & T \\
\bot & 1 & \bot & T \\
\top & \bot & d_1 \mathop{\hat{\text{op}}} d_2 & T \\
\top & T & T & T \\
\end{array}
\]
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></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>most optimistic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(bottom)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>most conservative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(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</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>$\bigcup$</td>
<td>$\bigcap$</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 \in S
f \in S
t \in \emptyset
u \in \land
Flow functions

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

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

\[ X := Y \]

\[ F_X := Y \text{(in)} = \]
Flow functions

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

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

```
x := read()
v := a + b

x := x + 1
w := x + 1

w := x + 1
a := w
v := a + b

z := x + 1
t := a + b
```
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: $(D, \sqsubseteq, \bot, T, \cup, \cap) =$
Example: live variables

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

\[
\begin{align*}
\text{in} & \quad X := Y \text{ op } Z \\
\text{out} & \quad F_X := Y \text{ op } Z(\text{out}) =
\end{align*}
\]
Example: live variables

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

\[
\begin{align*}
\text{in} & \quad \text{out} \\
X := Y \text{ op } Z & \quad F_X := Y \text{ op } Z(\text{out}) = \text{out} - \{X\} \cup \{Y, Z\}
\end{align*}
\]
Example: live variables

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

\[
x := x + 1
\]

\[
y := x + 10
\]

\[
\ldots y \ldots
\]
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

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

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

\[ \text{out} - \{x\} \cup x \notin \text{out} \? \emptyset : \{Y, Z\} \]
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
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;
...

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

if (...) {
  ... 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

```java
if (...) {
    x := -1;
} else {
    x := 1;
}
y := x * x;
```
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: $F_p = \overline{F_{s_1} \circ ... \circ F_{s_n}}$

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

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

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 $\text{MOP} \subseteq \text{dataflow}$ 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?

Dataflow

\[
x := -1; \\
x := 1; \\
\text{Merge} \\
y := x \times x; \\
\ldots \ y \ldots
\]

MOP

\[
x := -1; \\
x := 1; \\
y := x \times x; \\
\ldots \ y \ldots \\
\text{Merge}
\]
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:

\[ \forall a, b \cdot F(a \sqcup b) = F(a) \sqcup F(b) \]

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