Another example: constant prop

- Set D = $\mathcal{P}(\{ x \to c \mid x \in Vars, c \in Const \})$

```
     | in
 [ X := N ]
     | out
```

$F_{X := N}(in) = in \cup \{ x \to N \} - \{ x \to * \}$

```
       | in
 [ X := Y op Z ]
       | out
```

$F_{X := Y\ op\ Z}(in) = in - \{ x \to * \}$
$\cup \{ \quad \mid Y \to N$

Another example: constant prop

- Set D = $2^{\{ X \to N \mid X \in Vars \wedge N \in Z \}}$

a := 0
b := d/a;

```
     | in
 [ X := N ]
     | out
```

$F_{X := N}(in) = in - \{ X \to * \} \cup \{ X \to N \}$

```
       | in
 [ X := Y op Z ]
       | out
```

$F_{X := Y\ op\ Z}(in) = in - \{ X \to * \} \cup$
$\{ X \to N \mid (Y \to N_1) \in in \wedge$
$(Z \to N_2) \in in \wedge$
$N = N_1 \text{ op } N_2 \}$

Another example: constant prop

mapt (Y)

```
     | in
 [ X := *Y ]
     | out
```

$F_{X := *Y}(in) = in - \{ x \to * \}$
$\cup \{ x \to N \mid \forall v \in mapt(Y).\ v \to N \in in \}$

Z → N   Y → N

```
     | in
 [ *X := Y ]
     | out
```

$F_{*X := Y}(in) = in - \{ v \to N \mid v \in mapt(x) \}$

Z → N

Another example: constant prop

```
     | in
 [ X := *Y ]
     | out
```

$F_{X := *Y}(in) = in - \{ X \to * \}$
$\cup \{ X \to N \mid \forall Z \in \text{may-point-to}(Y) .$
$(Z \to N) \in in \}$

```
     | in
 [ *X := Y ]
     | out
```

$F_{*X := Y}(in) = in - \{ Z \to * \mid Z \in \text{may-point}(X) \}$
$\cup \{ Z \to N \mid Z \in \text{must-point-to}(X) \wedge$
$Y \to N \in in \}$
$\cup \{ Z \to N \mid (Y \to N) \in in \wedge$
$(Z \to N) \in in \}$

Another example: constant prop

```
       | in
 [ *X := *Y + *Z ]
       | out
```

$F_{*X := *Y + *Z}(in) = in -$

```
       | in
 [ X := G(...) ]
       | out
```

$F_{X := G(...)}(in) = \emptyset$

Another example: constant prop

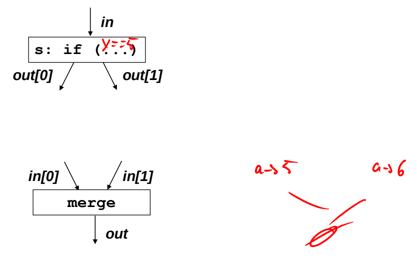
```
       | in
 [ *X := *Y + *Z ]
       | out
```

$F_{*X := *Y + *Z}(in) = F_{a := *Y; b := *Z; c := a + b; *X := c}(in$

```
       | in
 [ X := G(...) ]
       | out
```

$F_{X := G(...)}(in) = \emptyset$

Another example: constant prop



Lattice

FS = {x→c}

- (D, ⊑, ⊥, ⊤, ⊔, ⊓) =

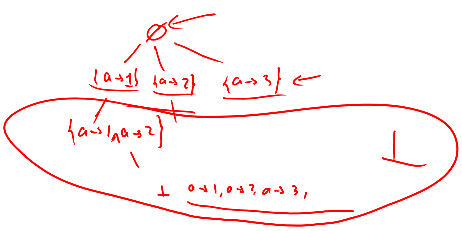
  ⊇ FS ∅ ∩ ∪

Lattice

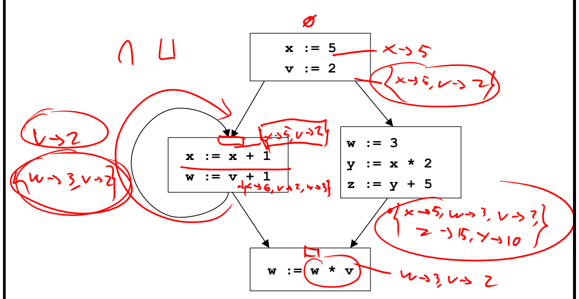
- (D, ⊑, ⊥, ⊤, ⊔, ⊓) =

  (2^A, ⊇, A, ∅, ∩, ∪)

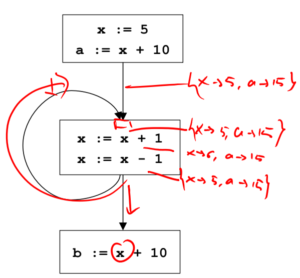
  where A = { x → N | x ∊ Vars ∧ N ∊ Z }



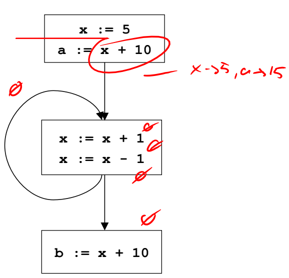
Example



Another Example



Another Example starting at top



Back to lattice

- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$
  $(2^A, \supseteq, A, \emptyset, \cap, \cup)$
  where $A = \{ x \to N \mid x \in \text{Vars} \wedge N \in Z \}$

- What's the problem with this lattice?

Back to lattice

- $(D, \sqsubseteq, \bot, \top, \sqcup, \sqcap) =$
  $(2^A, \supseteq, A, \emptyset, \cap, \cup)$
  where $A = \{ x \to N \mid x \in \text{Vars} \wedge N \in Z \}$

- What's the problem with this lattice?

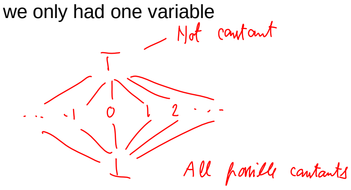
- Lattice is infinitely high, which means we can't guarantee termination

Better lattice

- Suppose we only had one variable

Better lattice

- Suppose we only had one variable



- $D = \{\bot, \top\} \cup Z$
- $\forall i \in Z . \bot \sqsubseteq i \wedge i \sqsubseteq \top$
- height = 3

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
  $\bot = (\bot_1, \ldots, \bot_n)$
  $\top = (\top_1, \ldots, \top_n)$
  $(a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup_1 b_1, \ldots, a_n \sqcup_n b_n)$
  $(a_1, \ldots, a_n) \sqcap (b_1, \ldots, b_n) = (a_1 \sqcap_1 b_1, \ldots, a_n \sqcap_n b_n)$
  height = height(D_1) + ... + height(D_n)

For all variables

- Option 2: Map from variables to single lattice
- Given lattice $(D, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and a set V, create:

map lattice $V \to D = (V \to D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$

For all variables

- Option 2: Map from variables to single lattice
- Given lattice $(D, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and a set V, create:

map lattice $V \to D = (V \to D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$

$\bot = \lambda v. \bot_1$

$\top = \lambda v. \top_1$

$a \sqcup b = \lambda v. (a(v) \sqcup_1 b(v))$

$a \sqcap b = \lambda v. (a(v) \sqcap_1 b(v))$

$a \sqsubseteq b \Leftrightarrow \forall v. a(v) \sqsubseteq_1 b(v)$

Back to example

in

```
X := Y op Z
```

out

in(Y) op in(Z)

$F_{X := Y \text{ op } Z}(\text{in}) = \text{in}[X \mapsto \text{in}(Y) \ \widehat{op} \ \text{in}(Z)]$

$\sqsubseteq \quad \widehat{op} \quad \sqsubseteq$

$\widehat{op}$	$\bot$	c_1	$\top$
$\bot$	$\bot$	$\top$	$\top$
c_2	$\bot$	$c_1 \text{ op } c_2$	$\top$
$\top$	$\top$	$\top$	$\top$

Back to example

in

```
X := Y op Z
```

out

$F_{X := Y \text{ op } Z}(\text{in}) = \text{in}[\, X \to \text{in}(Y) \ \widehat{op} \ \text{in}(Z) \,]$

where $a \ \widehat{op} \ b$ =

$\widehat{op}$	$\bot$	d_1	$\top$
$\bot$	$\bot$	$\bot$	$\top$
d_2	$\bot$	$d_1 \text{ op } d_2$	$\top$
$\top$	$\top$	$\top$	$\top$

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 $x \to y$ must-point-to pairs
  - if we compute $x \to y$, then, then during program execution, x must point to y
- Set of $x \to y$ may-point-to pairs
  - if during program execution, it is possible for x to point to y, then we must compute $x \to y$

$A \Rightarrow B$

$\neg B \Rightarrow \neg A$

May vs must

	May	Must
most optimistic (bottom)	$\emptyset$	FS
most conservative (top)	FS	$\emptyset$
safe	add	smaller
merge	$\cup$	$\cap$

May vs must

	May	Must
most optimistic (bottom)	empty set	full set
most conservative (top)	full set	empty set
safe	overly big	overly small
merge	$\cup$	$\cap$

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:

$S = \{ X \to E \mid X \in Var, E \in Expr \}$

$D = 2^S$

$\bot = S$

$\top = \emptyset$

$\sqcup = \cap$

Flow functions

$a \to b + c$

```
   | in
X := Y op Z
   | out
```

$F_{X := Y \text{ op } Z}(in) = in - \{ X \to * \} - \{ * \to \ldots X \ldots \} \cup \{ X \to Y \text{ op } Z \}$

$X \neq Y \wedge Y \neq Z$

```
   | in
 X := Y
   | out
```

$F_{X := Y}(in) =$

Flow functions

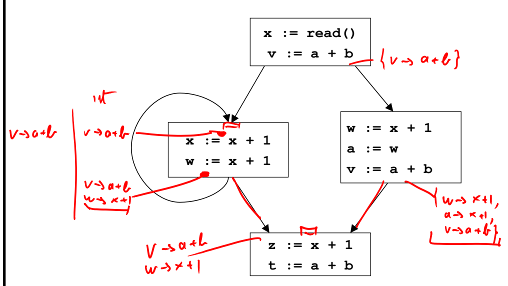
```
   | in
X := Y op Z
   | out
```

$F_{X := Y \text{ op } Z}(in) = in - \{ X \to * \}$
$- \{ * \to \ldots X \ldots \} \cup$
$\{ X \to Y \text{ op } Z \mid X \neq Y \wedge X \neq Z \}$

```
   | in
 X := Y
   | out
```

$F_{X := Y}(in) = in - \{ X \to * \}$
$- \{ * \to \ldots X \ldots \} \cup$
$\{ X \to E \mid Y \to E \in in \}$

Example

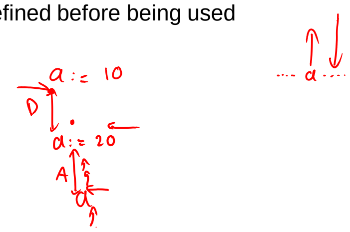


Direction of analysis

- 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

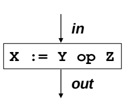


Example: live variables

- Set D = $\mathcal{P}$(Vars)
- Lattice: (D, ⊑, ⊥, ⊤, ⊔, ⊓) =
  ∅ Vars ∪ ∩

Example: live variables

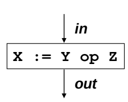
- Set D = 2 $^{\text{Vars}}$
- Lattice: (D, ⊑, ⊥, ⊤, ⊔, ⊓) = (2$^{\text{Vars}}$, ⊆, ∅ ,Vars, ∪, ∩)



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

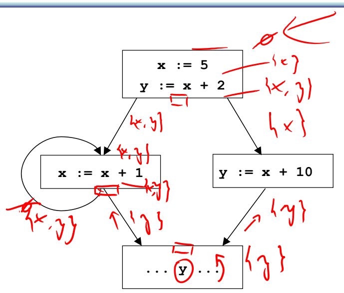
Example: live variables

- Set D = 2 $^{\text{Vars}}$
- Lattice: (D, ⊑, ⊥, ⊤, ⊔, ⊓) = (2$^{\text{Vars}}$, ⊆, ∅ ,Vars, ∪, ∩)



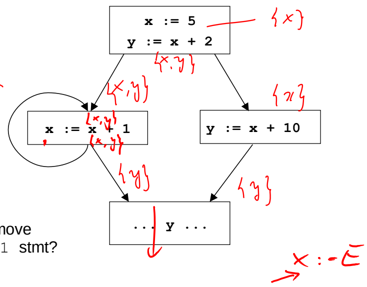
$F_{X := Y \text{ op } Z}(\text{out}) = \text{out} - \{ X \} \cup \{ Y, Z\}$

Example: live variables



Example: live variables

1st iter {y} | 2nd iter {x, y}



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

x := E

Revisiting assignment

b := e + f
a := b + c

in

~~X := Y op Z~~

out

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

X ∉ out ? out : (out - {X} ∪ {Y, Z})

Y := X + 1
X := X + 1
{y}

Revisiting assignment

in

X := Y op Z

out

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

out - {X} ∪

X ∉ out ? ∅ : {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?

```
if (p) {
  a := 7
} else {
  a := 4
}
```

```
if (p) {
  b := a - 2
} else {
  b := a + 1
}
b := 5
```

Precision

- Going back to constant prop, in what cases would we lose precision?

```
x := 5
if (<expr>) {
    x := 6
}
... x ...

where <expr> is
equiv to false
```

```
if (p) {
    x := 5;
} else
    x := 4;
}
...
if (p) {
    y := x + 1
} else {
    y := x + 2
}
... y ...
```

```
if (...) {
    x := -1;
} else
    x := 1;
}
y := x * x;
... 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

```
if (...) {
    x := -1;
} else
    x := 1;
}
y := x * x;
... y ...
```

x:=-1 y:=x*x y→1 x:=1 y:=x*x y→1 y→1 ⊔

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 = $F_{s_1} \circ \cdots \circ F_{s_n}$
- Given an edge e, let paths-to(e) be the (possibly infinite) set of paths that lead to e
- Given an edge e, MOP(e) = $\bigsqcup_{p \in \text{paths-to}(e)} F_p(\bot)$
- 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 ⊑ 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

$a*(b+c) = a*b + a*c$

- 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;
       Merge
   y := x * x;
   ... y ...
```

a, b, ⊔, a ⊔ b, F, F(a ⊔ b), y→⊤

MOP

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

a, F(a), b, F(b), F(a) ⊔ F(b), y→1

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