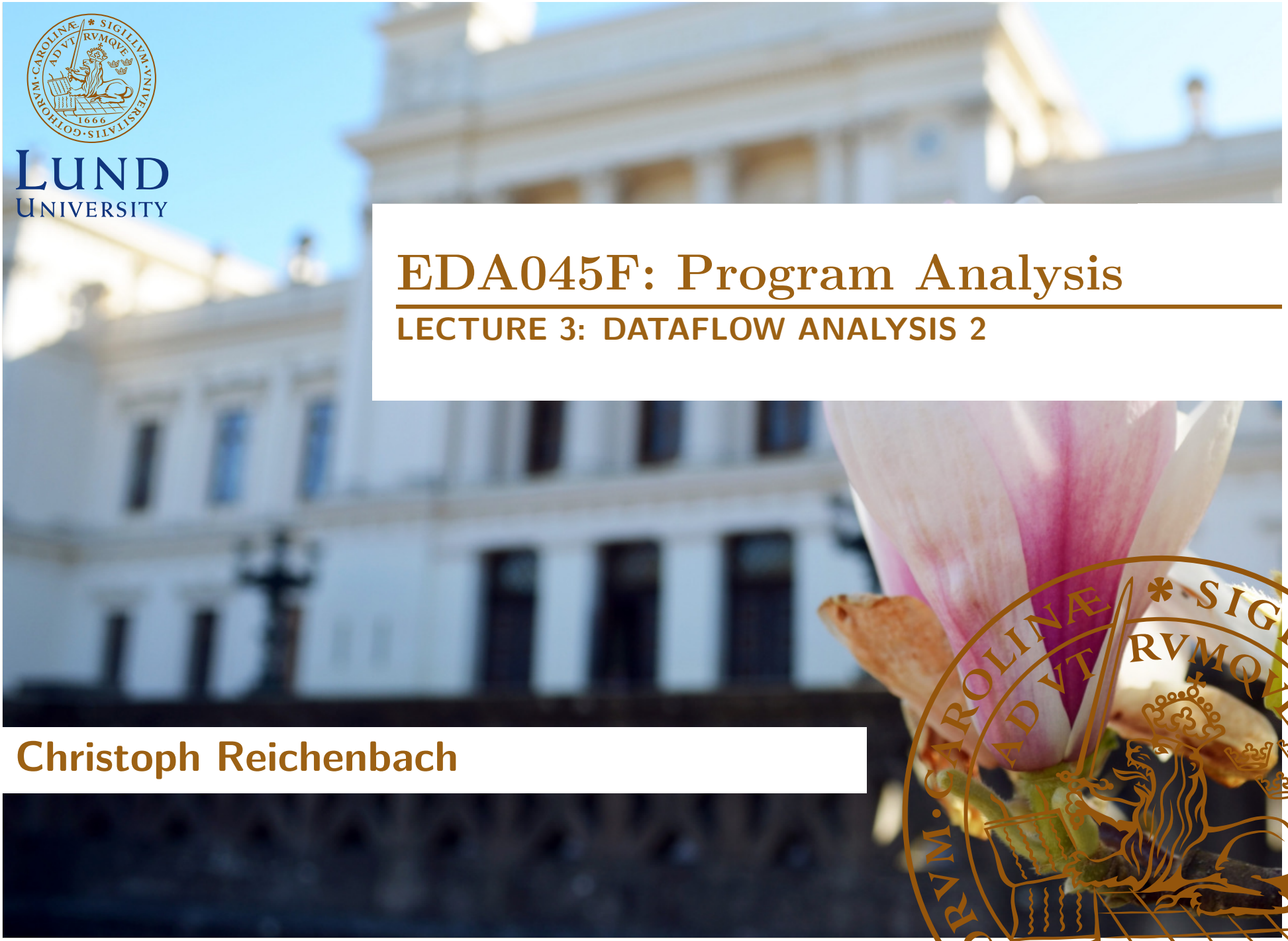


EDA045F: Program Analysis

LECTURE 3: DATAFLOW ANALYSIS 2

Christoph Reichenbach

In the last lecture...

- Eliminating Nested Expressions (Three-Address Code)
- Control-Flow Graphs
- Static Single Assignment Form
- Basic Dataflow Analysis
  - $trans_b(x)$
  - $merge_b(x, y)$
- Reaching Definitions Analysis
- Live Variables Analysis

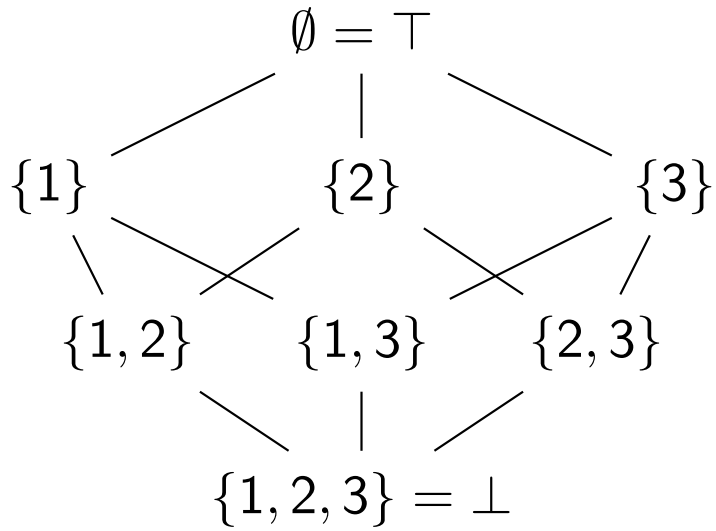
Dataflow Analysis

Analyse properties of variables or basic blocks

Examples in practice:

- *Live Variables*
  Is this variable ever read?
- *Reaching Definitions*
  What are the possible values for this variable?
- *Available Expressions*
  What variable definitely has which expression?

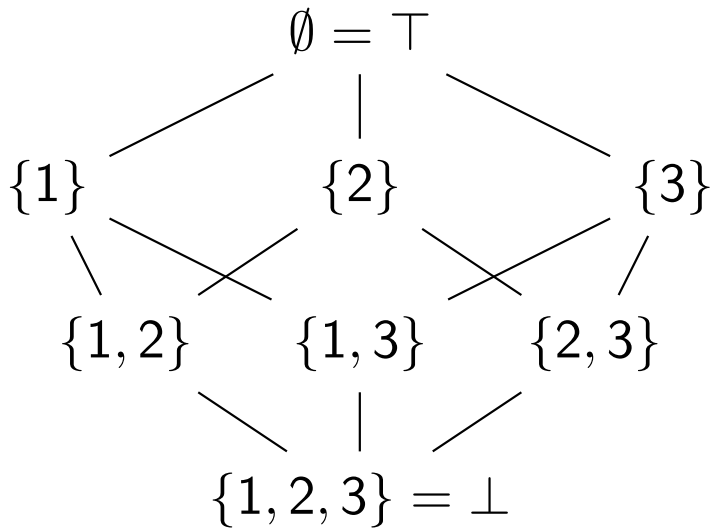
Analyses on Powersets (1/2)



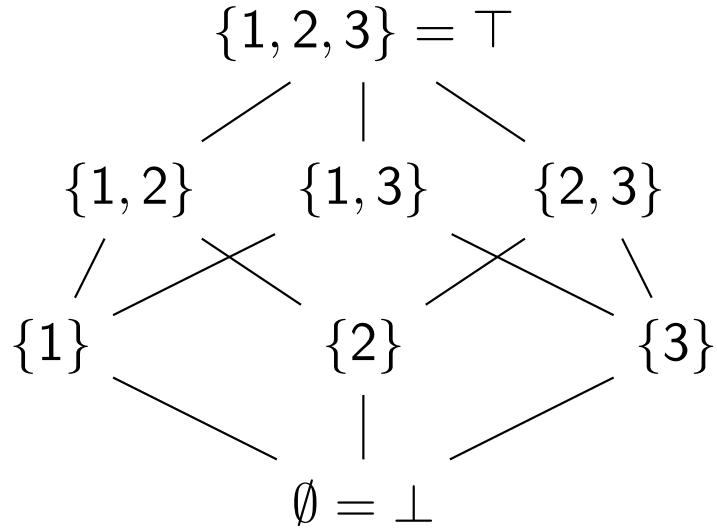
$merge_b = \cup$

- Common: 'Which elements of S are possible / necessary?'
  - $S \subseteq \mathbb{Z}$ (*Reaching Definitions*)
    - $S =$ Numeric Constants in code $\cup \{0, 1\}$
  - $S =$ Variables (*Live Variables*)
  - $S =$ Program Locations (*alt. Reaching Definitions*)
  - $S =$ Types
- Abstract Domain: Powerset $\mathcal{P}(S)$
  - Finite iff S is finite

Analyses on Powersets (2/2)



$merge_b = \cup$



$merge_b = \cap$

- $merge_b$ can be $\cup$ or $\cap$
- $\cup$:
  - Property that is true over *any* path
  - **May**-analysis (e.g., Reaching Definitions)
- $\cap$:
  - Property that is true over *all* paths
  - **Must**-analysis

Gen-Sets and Kill-Sets

- Many transfer functions $trans_b$ have the following form:
  - Remove set of options $kill_{x,b}$ from each variable x
  - Add set of options $gen_{x,b}$ to each variable x
  - Don't depend on other variables

$$trans_b(\{x \mapsto A, \ldots\}) = \{x \mapsto (A \setminus kill_{x,b}) \cup gen_{x,b}, \ldots\}$$

- Highly efficient implementation with bit-vectors possible
- Examples:
  - *Reaching Definitions* on finite domain
    - *gen*: assignments in current basic block
    - *kill*: everything else if variable is assigned
  - *Live Variables*
    - *gen*: used variables
    - *kill*: overwritten variables

Gen/Kill: Available Expressions

"Which expressions do we currently have evaluated and stored?"

C

```c
int x = 3 + z;
int y = 2 + z;
if (z > 0) {
  x = 4;
}
f(2 + z); // Can re-use y here!
```

- Forward analysis
- *gen*: any expression assigned to the variable
- *kill*: any other expression
- $merge_b = \cap$

Gen/Kill: Very Busy Expressions

"Which expression do we definitely need to evaluate at least once?"

C

```
// (x / 42) is very busy:  (A),(B)
if (z > 0) {
  x = 4 + x / 42; // (A)
  y = 1;
} else {
  x = x / 42; // (B)
}
g(x);
```

- Backwards analysis
- *gen*: any expression assigned to the variable
- *kill*: any other expression
- $merge_b = \cap$

Summary

- Common: Abstract Domain is powerset of some set S
- Transfer function $trans_b$:

$$trans_b(\{x \mapsto A, \ldots\}) = \{x \mapsto (A \setminus kill_{x,b}) \cup gen_{x,b}, \ldots\}$$

- *kill*: 'Kill set': Entries of S to remove
- *gen*: 'Gen set': Entries of S to add
- $merge_b$ is $\cup$ or $\cap$
- Often admits very efficient implementation

	May	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Lattices: Models for Information

- Program analyses model *information*
- Undecidability $\Longrightarrow$ must approximate
  - **Conservative:** over-approximate (*contradictory information*)
  - **Optimistic:** under-approximate (*incomplete information*)
- Commonly used formal model: *lattices*

Partial Ordering

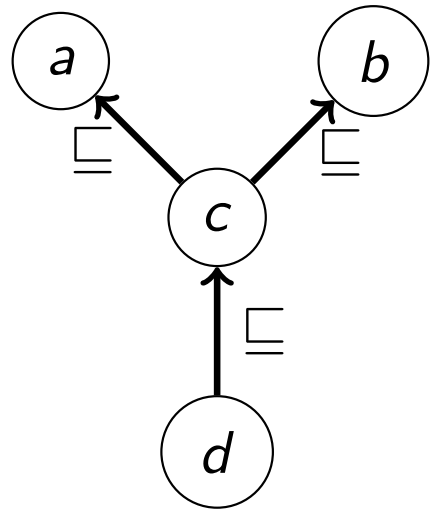
Lattices L are based on a *partially ordered set* $\langle \mathcal{L}, \sqsubseteq \rangle$:

- Set: $\mathcal{L}$ describes possible information
- $(\sqsubseteq) \subseteq \mathcal{L} \times \mathcal{L}$:
- Intuition for $a \sqsubseteq b$ (for program analysis):
  - a has at least as much information as b
- $(\sqsubseteq)$ is a *partial order*:

$$
\begin{array}{lll}
a \sqsubseteq a & & \textit{Reflexivity} \\
a \sqsubseteq b \text{ and } b \sqsubseteq a & \Longrightarrow \quad a = b & \textit{Antisymmetry} \\
a \sqsubseteq b \text{ and } b \sqsubseteq c & \Longrightarrow \quad a \sqsubseteq c & \textit{Transitivity}
\end{array}
$$

- Example:
  - $\mathcal{L} = \{\textit{unknown}, \textit{true}, \textit{false}, \textit{true-or-false}\}$
  - $\textit{true-or-false} \sqsubseteq \textit{true} \sqsubseteq \textit{unknown}$
  - $\textit{true-or-false} \sqsubseteq \textit{false} \sqsubseteq \textit{unknown}$

Greatest Lower bound



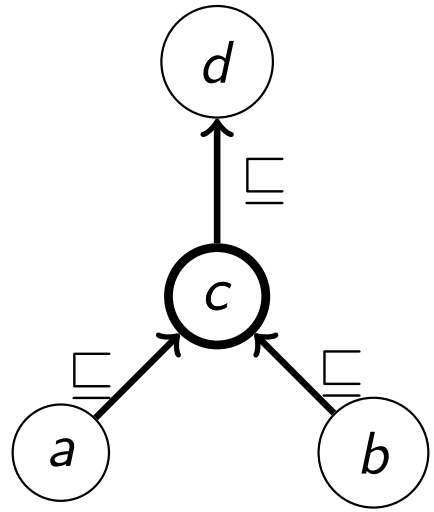
Combining potentially contradictory information:

- *Meet operator*: $(\sqcap) : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$
- $a \sqcap b \sqsubseteq a$ and $a \sqcap b \sqsubseteq b$
- *Greatest* element with this property:

$$\text{for all } d : d \sqsubseteq a \text{ and } d \sqsubseteq b \implies d \sqsubseteq a \sqcap b$$

$\sqcap$ computes *Greatest Lower Bound* (Infimum)

Least Upper Bound



Converse operation:

- *Join operator*: $(\sqcup) : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$
- $a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$
- *Least* element with this property:

$$\text{for all } d : a \sqsubseteq d \text{ and } a \sqsubseteq d \implies d \sqsupseteq a \sqcup b$$

$\sqcup$ computes *Least Upper Bound* (Supremum)

Lattices

$$L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup \rangle$$

- $\mathcal{L}$: Underlying set
- $(\sqsubseteq) \subseteq \mathcal{L} \times \mathcal{L}$: Partial Order
- $(\sqcap) : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$: Meet (computes g.l.b.)
- $(\sqcup) : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$: Join (computes l.u.b.)
- Join/Meet always exist and are unique
- It can be shown that $(\sqcup)$, $(\sqcap)$ are:

$$\begin{aligned} \text{Commutative:} \quad & a \sqcap b = b \sqcap a \\ \text{Associative:} \quad & a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \end{aligned}$$

(Analogous for $\sqcup$)

Complete Lattices

A lattice $L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup \rangle$ is *complete* iff:

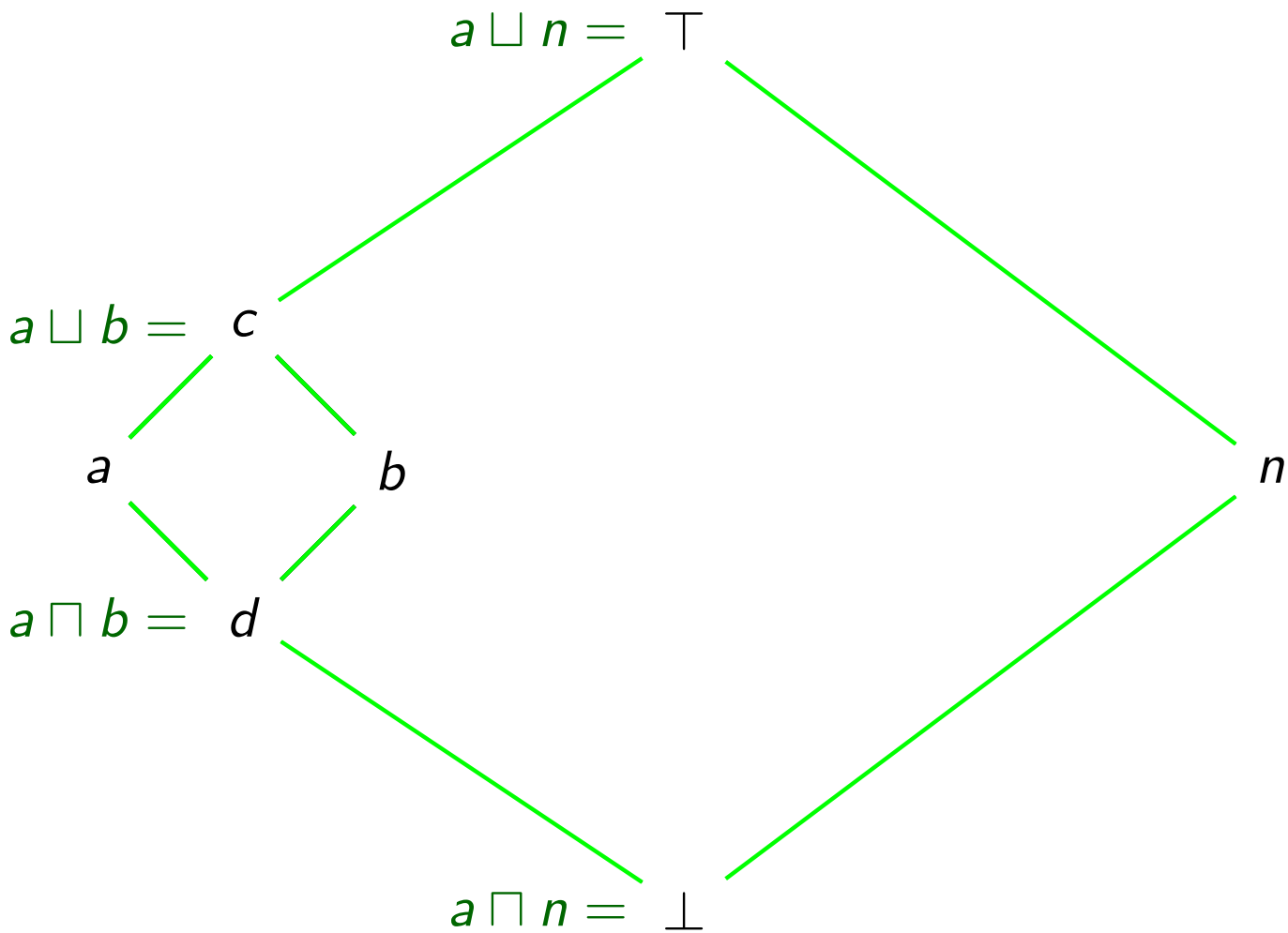
- For any $\mathcal{L}' \subseteq \mathcal{L}$ there exist:
  - $\bigsqcup \mathcal{L}'$ (least upper bound for arbitrary set)
  - $\bigsqcap \mathcal{L}'$ (greatest lower bound for arbitrary set)
- We define $\top \sqsupseteq a$ for all $a \in \mathcal{L}$ as:

$$\top = \bigsqcup \mathcal{L}$$

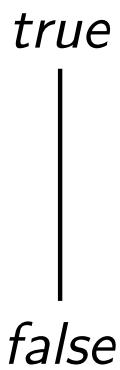
- We define $\bot \sqsubseteq a$ for all $a \in \mathcal{L}$ as:

$$\bot = \bigsqcap \mathcal{L}$$

Complete Lattices: Visually

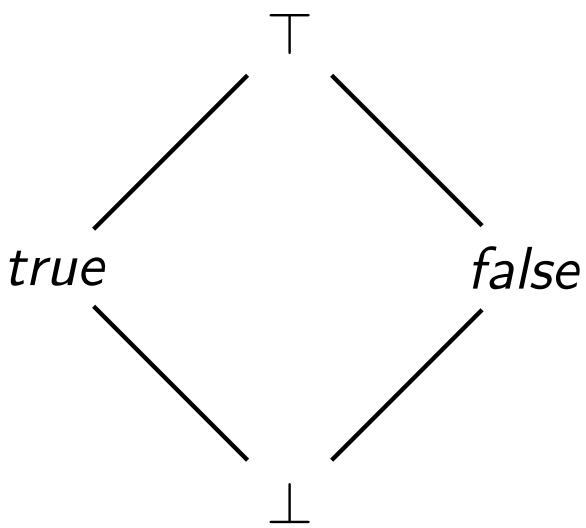


Example: Binary Lattice



- $\top$ = *true*
- $\bot$ = *false*
- $\sqcup$ = *logical "or"*
- $\sqcap$ = *logical "and"*

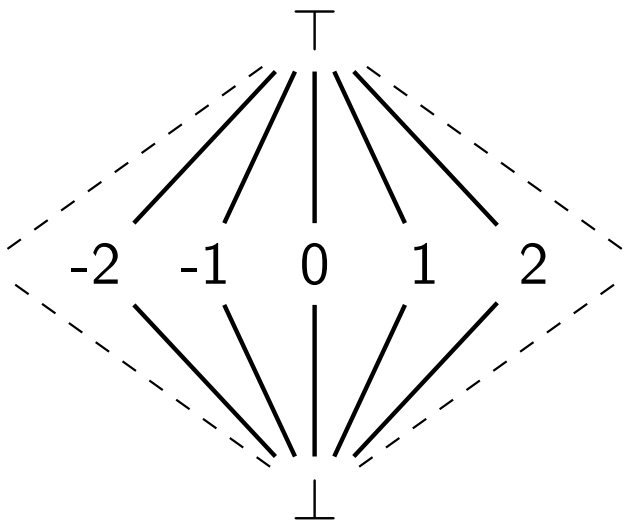
Example: Booleans



- $\top$ = true-and-false
- $\bot$ = true-or-false
- $a \sqcup b$: must be both a and b
- $a \sqcap b$: could be either a or b
- If $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$:
  - Lattice sometimes called $\mathbb{B}_{\bot}^{\top}$

Other interpretations possible

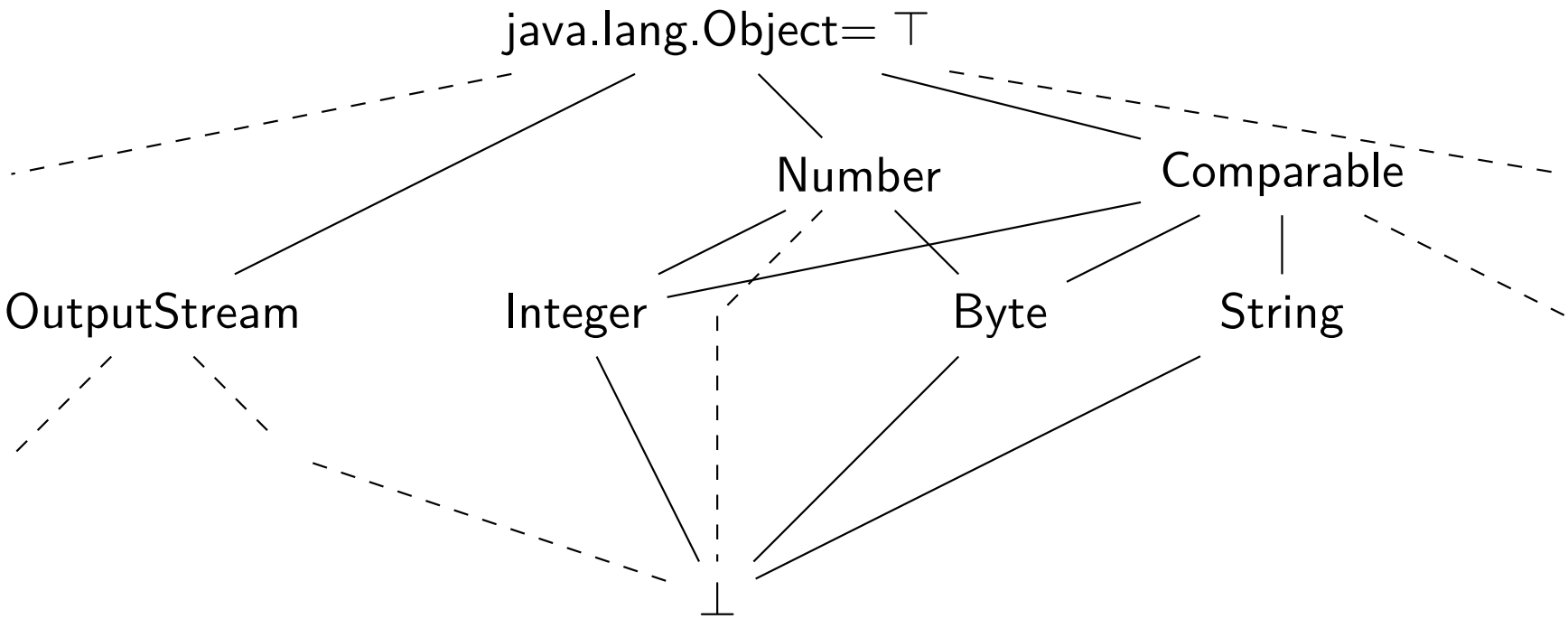
Example: Flat Lattice on Integers



- Sometimes written $\mathbb{Z}_{\perp}^{\top}$
- $\top = \emptyset$
- $\perp = \mathbb{Z}$
- $a \sqcup b = \begin{cases} a & \text{iff } a = b \\ \top & \text{otherwise} \end{cases}$
- $a \sqcap b = \begin{cases} a & \text{iff } a = b \\ \perp & \text{otherwise} \end{cases}$

Analogous for other $X_{\perp}^{\top}$ from set X

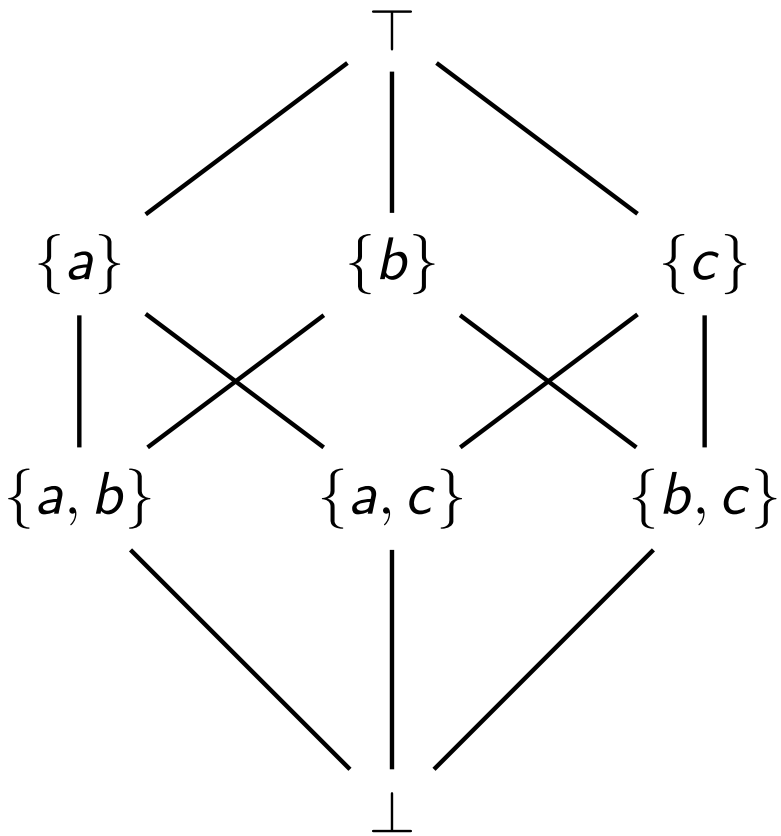
Example: Type Hierarchy Lattices



- Type systems with subtyping form (non-powerset) lattice
- Must add $\bot$ element
- Some langugaes (C++) need extra $\top$ element
- For extra precision, we may add nodes for e.g.

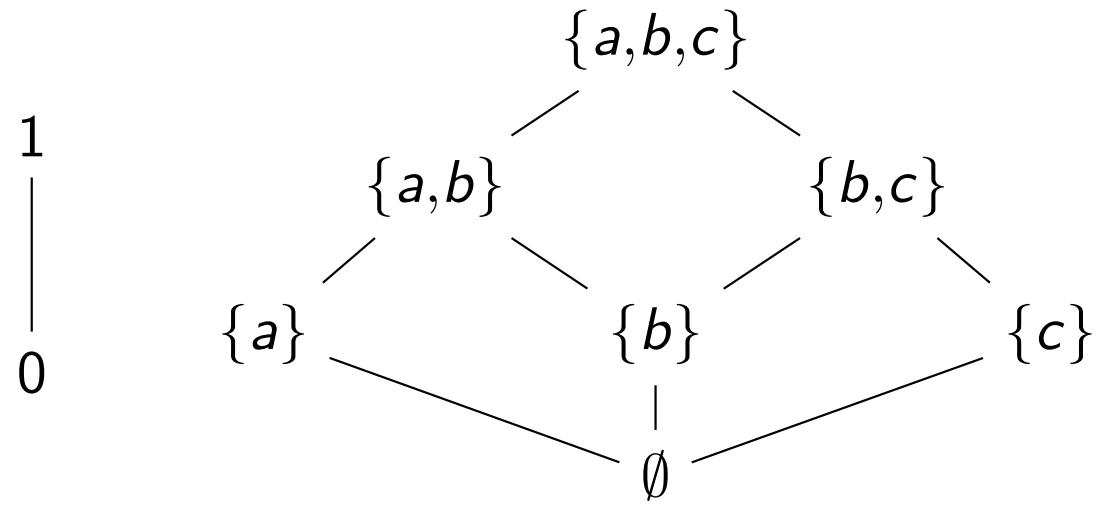
$$\text{java.lang.Comparable} \sqcap \text{java.lang.Serializable}$$

Example: Powersets



- Take any set $S = \{a, b, c\}$
- $\mathcal{L} = \mathcal{P}(S)$
- $\top = \emptyset$
- $\bot = S$
- $(\sqcup) = (\cap)$
- $(\sqcap) = (\cup)$

Example: Lattices and Non-Lattices



Lattice **Lattice** **Not A Lattice**

Right-hand side is missing e.g. a unique $R \sqcup S$

Example: Natural numbers with 0, ω

- $\top = \omega$
- $\bot = 0$
- $a \sqcup b =$ maximum of a and b
- $a \sqcap b =$ minimum of a and b

Dual Lattices

Let $L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup \rangle$ be a lattice. Then:

- $\overline{L} = \langle \mathcal{L}, \sqsupseteq, \sqcup, \sqcap \rangle$ is *also* a lattice
- $\overline{L}$ is *dual lattice* to L
- If L is complete, with $\top_L$, $\bot_L$ being top, bottom:
  - $\overline{L}$ is also complete, and:
  - $\top_{\overline{L}} = \bot_L$
  - $\bot_{\overline{L}} = \top_L$

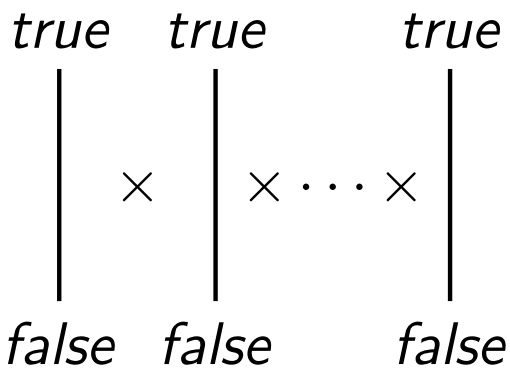
Lattices can be 'flipped around' without losing their properties

Product Lattices

- Program analysis: Each variable needs its own lattice
- Can we combine these lattices to analyse variables simultaneously?
- Assume (complete) lattices:
  - $L_1 = \langle \mathcal{L}_1, \sqsubseteq_1, \sqcap_1, \sqcup_1, \top_1, \bot_1 \rangle$
  - $L_2 = \langle \mathcal{L}_2, \sqsubseteq_2, \sqcap_2, \sqcup_2, \top_2, \bot_2 \rangle$
- Let $L_1 \times L_2 = \langle \mathcal{L}_1 \times \mathcal{L}_2, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$ where:
  - $\langle a, b \rangle \sqsubseteq \langle a', b' \rangle$ iff $a \sqsubseteq_1 a'$ and $b \sqsubseteq_2 b'$
  - $\langle a, b \rangle \sqcap \langle a', b' \rangle = \langle a \sqcap_1 a', b \sqcap_2 b' \rangle$
  - $\langle a, b \rangle \sqcup \langle a', b' \rangle = \langle a \sqcup_1 a', b \sqcup_2 b' \rangle$
  - $\top = \langle \top_1, \top_2 \rangle$
  - $\bot = \langle \bot_1, \bot_2 \rangle$

Point-wise products of (complete) lattices are again (complete) lattices

Product Lattices over Binary Lattices



- Recall binary lattices:
  - $\top$ = *true*
  - $\bot$ = *false*
  - $\sqcup$ = *logical "or"*
  - $\sqcap$ = *logical "and"*
- Computer hardware can compute $\sqcup$, $\sqcap$ of multiple lattices in parallel:
  - Bitwise or/and

  $\Longrightarrow$ Highly efficient
- Can represent other lattices efficiently, too

Give rise to highly efficient *Gen-/Kill-Set* based program analysis

Omitted Formal Details

What we didn't cover:

- Partially Ordered Sets (Posets)
- Semi-lattices (lacking meet or join)
- Lattice absorption laws
- Many interesting lattice properties

Not a full introduction to lattice theory

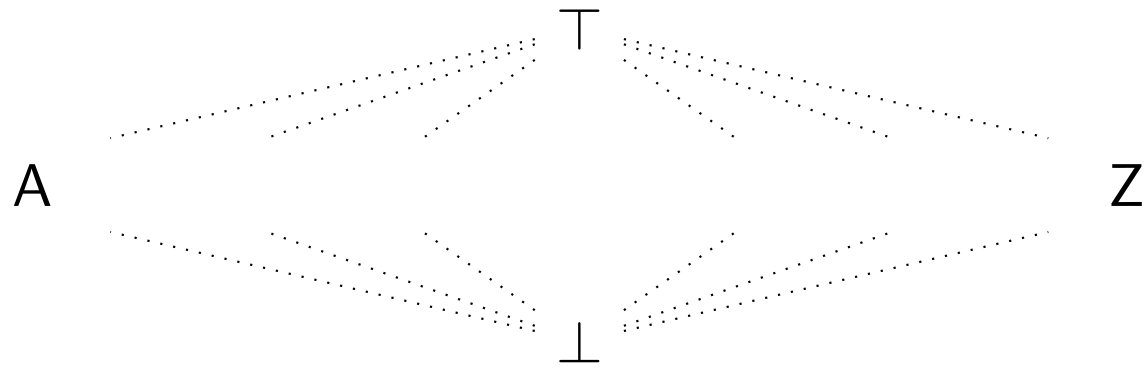
Word of Caution

- Definition in the book flips lattices
- $\top$ and $\bot$ mean the opposite
- We use the more common definition from the research literature
- Definition is *isomorphic*, but can be confusing...

Summary

- Complete lattices are formal basis for many program analyses
- Complete lattice $L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$
  - $\mathcal{L}$: Carrier set
  - ($\sqsubseteq$): Partial order
  - ($\sqcap$): Meet operation: find greatest lower bound
    (Analysis: $merge_b$)
  - ($\sqcup$): Join operation: find least upper bound
    (Analysis: uncommon, but can improve precision of two conservative results)
  - $\top$: Top-most element of complete lattice
    (Analysis: 'I don't know')
  - $\bot$: Bottom-most element of complete lattice
    (Analysis: 'I know that I can't know')
- Lattices can be flipped
- Lattices can be combined into *product lattices*

Monotone Frameworks



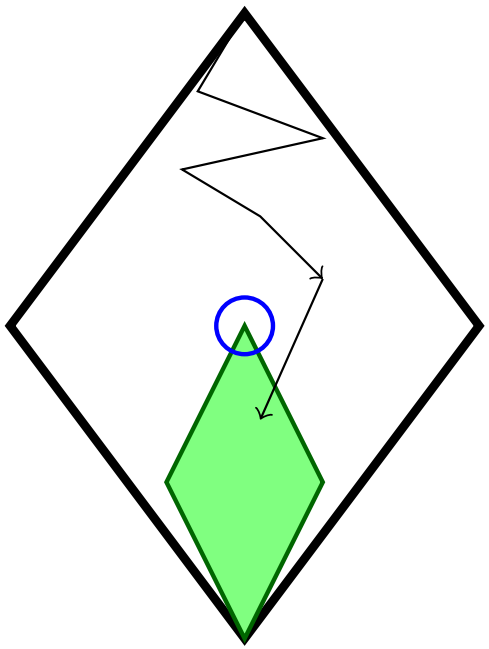
- $L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup \rangle$ is *complete lattice*
  - Order: $\sqsubseteq$
  - $\top$: 'No information (yet)'
  - $\bot$: 'Too much information / could be anything'
  - $merge_b(x, y) = x \sqcap y$ for all b
- $trans_b$ *monotonic*: $x \sqsubseteq y \implies trans_b(x) \sqsubseteq trans_b(y)$
- Finite lattice height ($\iff$ *Descending Chain Condition*):

  $x_0 \sqsupseteq x_1 \sqsupseteq x_2 \sqsupseteq x_3 \ldots \implies$ exists k s.th. for all $k' > k, x_{k'} = x_k$
- $\mathcal{L}$ can be infinite– only the lattice *height* must be finite

Putting It All Together

- Monotone Frameworks ensure termination of Data Flow Analysis
  - Information from complete Lattice L:
    - $\top$: 'No information (yet)'
    - $\bot$: 'Too much information / could be anything'
  - Must satisfy Descending Chain Condition: *no infinite progress*
- Ensure that no analysis step loses knowledge:
  - Each basic block has transfer function $trans_b$
    - Output knowledge $\mathbf{out}_b$ from input knowledge $\mathbf{in}_b$
    - *Monotonic*: increasing input knowledge does not decrease output knowledge
  - Merging multiple inputs with $merge_b = \sqcap$ is lattice meet (greatest lower bound)

Fixpoints



- Algorithm sketch from last week:
  - Repeat $trans_b$ and $merge_b$ until value no longer moves
  - Fixpoint
- Multiple possible solutions, ordered by $\sqsubseteq$
- Maximal Fixpoint $\Rightarrow$ Highest Precision

Value Range Analysis

'Find *value range* (interval of possible values) for x'

Python

```
x = 1
while ...:
  if ...:
    x = 4
  else:
    x = 7
```

- Multiple possible *sound* solutions:
  - $[1, 7]$
  - $[1, 10]$
  - $[-99, 99]$
  - $\bot$
- All of these values are fixpoints
- $[1, 7]$ is *maximum fispoint*

An Algorithm for MFP

- Last week: sketched naive algorithm for computing fixpoint
  - Produces maximum fixpoint (MFP)
- Optimise processing with *worklist*
  - Set-like datastructure:
    - **add** element (if not already present)
    - **contains** check: is element present?
    - **pop** element: remove and return one element
  - Tracks *what's left to be done*

The MFP Algorithm

```
Procedure MFP(⊤, merge_, ⊑, CFG, trans_, is-backward):
begin
  if is-backward then reverse edges(CFG);
  worklist := edges(CFG); -- edges that we need to look at
  foreach n ∈ nodes(CFG) do
    analysis[n] = ⊤;      -- state of the analysis
  done
  while not empty(worklist) do
    ⟨n, n'⟩ := pop(worklist);
    if analysis[n'] ⊒ trans_n(analysis[n]) then begin
      analysis[n'] := merge_n(analysis[n'], trans_n(analysis[n]))
      foreach n'' ∈ successor-nodes(CFG, n') do
        push(worklist, ⟨n', n''⟩);
      done
    end
  done
  return analysis;
end
```

Worklist allows focussing effort!

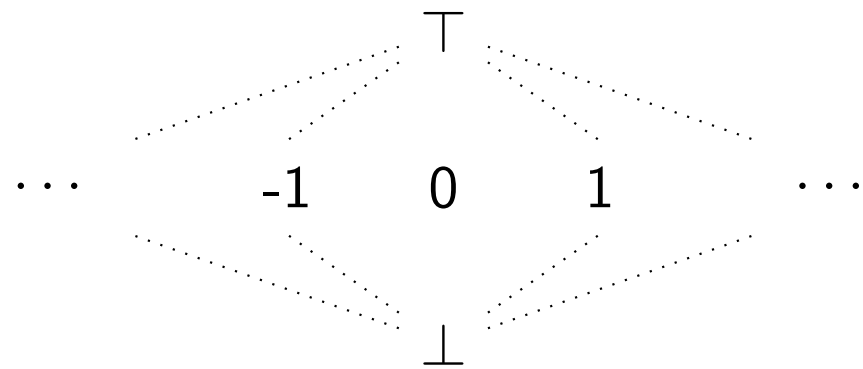
Summary: MFP Algorithm

- Compute data flow analysis:
  - Initialise all nodes with $\top$
  - Repeat until nothing changes any more:
    - Apply transfer function
    - Propagate changes along control flow graph
    - Apply $\sqcap$
- Compute maximal **fixpoint**
- Use **worklist** to increase efficiency
- Distinction: Forward/Backward analyses

Another Dataflow Example

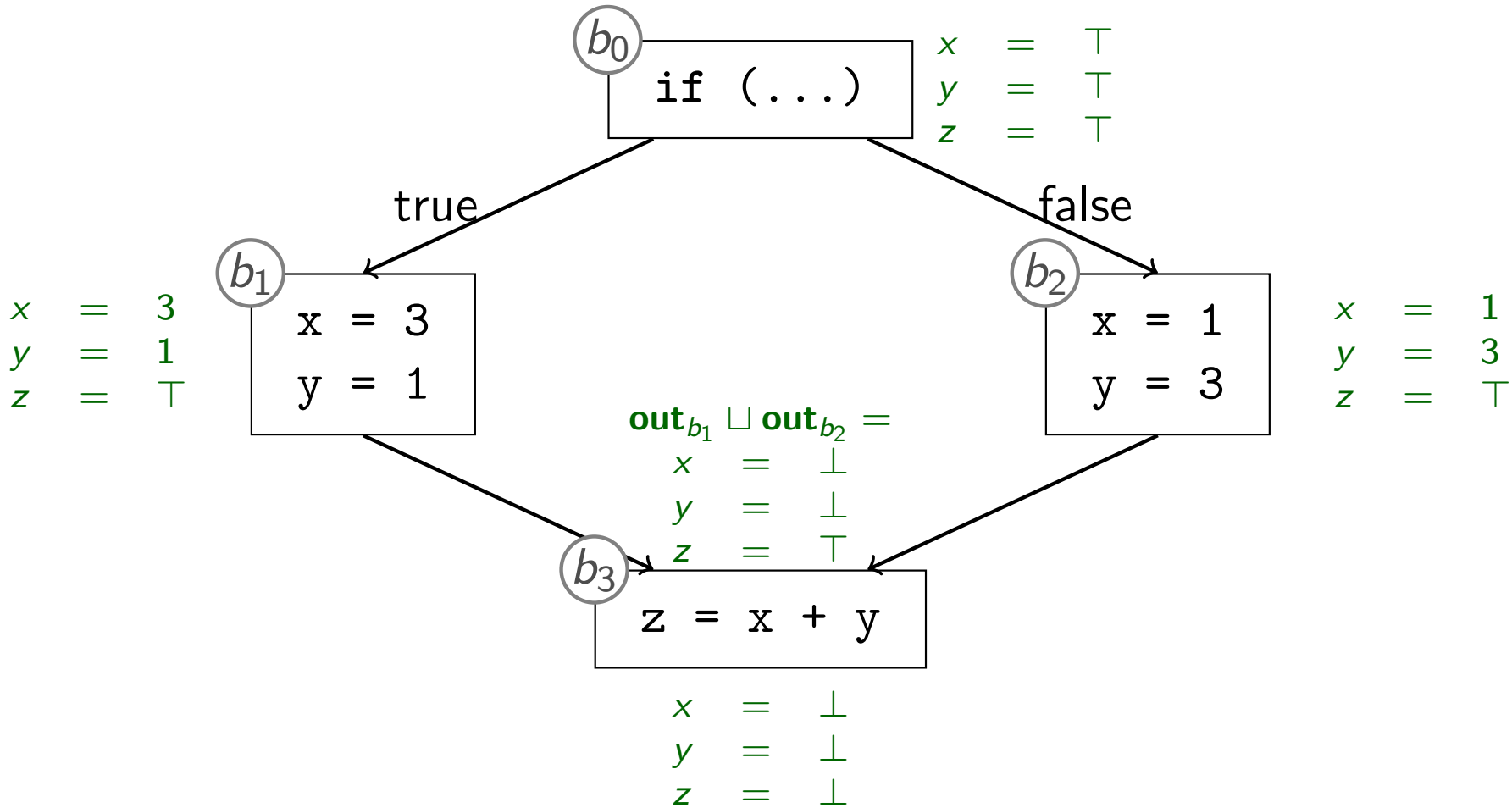
Consider again **Reaching Definitions**:

Lattice:



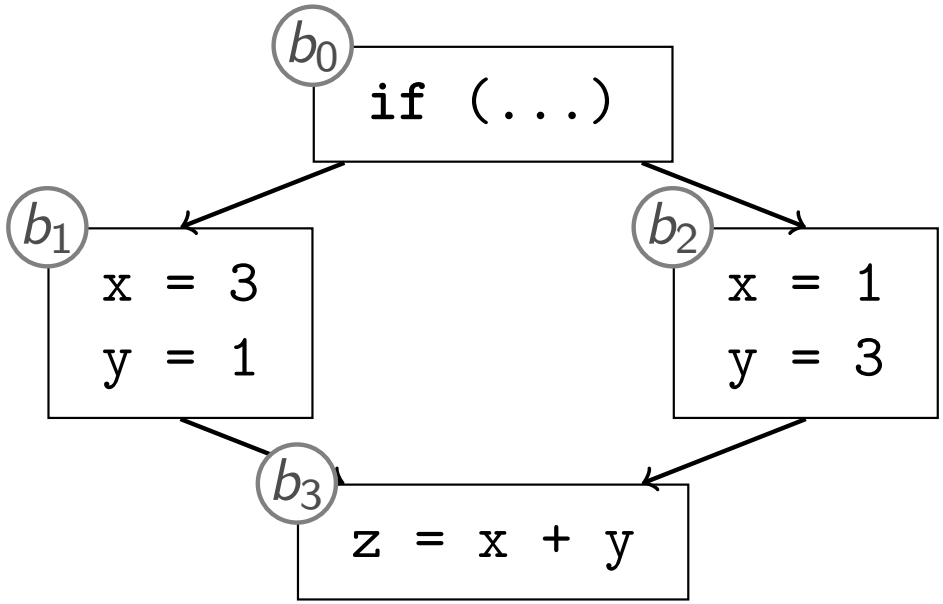
- $\bot$: Unknown
- $\top$: Too much/contradictory information
- Integer: exactly that one assignment is possible

Optimal Dataflow Results



Imprecise! Can we do better?

Execution paths



- Idea: Let's consider all *paths* through the program:

$$
\begin{aligned}
path_{b_0} &= \{[]\} \\
path_{b_1} &= \{[b_0]\} \\
path_{b_2} &= \{[b_0]\} \\
path_{b_3} &= \{[b_0, b_1], [b_0, b_2]\}
\end{aligned}
$$

The MOP algorithm for Dataflow Analysis

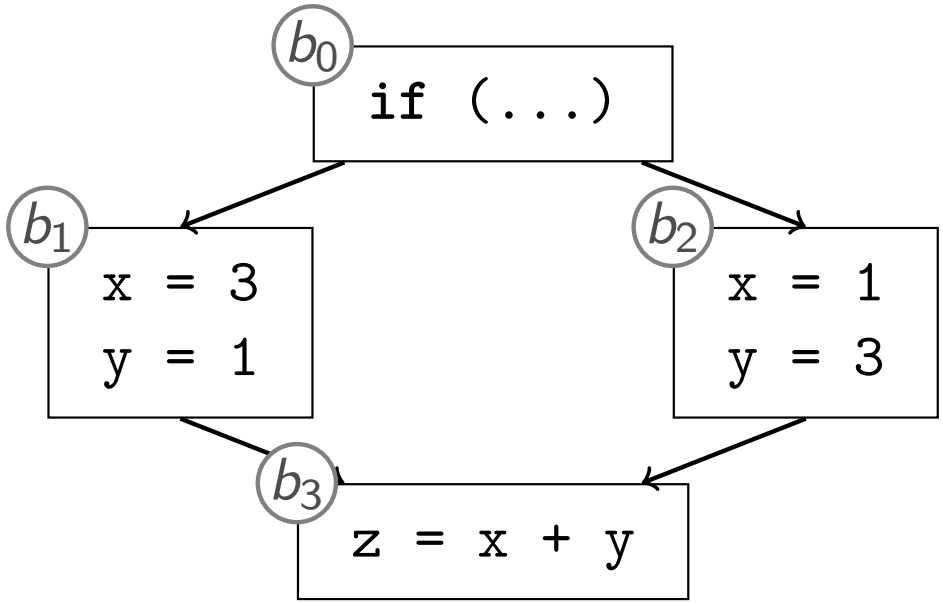
- Compute the MOP ('meet-over-all-paths') solution:
  - Iterate over all paths $[p_0, \ldots, p_k]$ in $path_{b_i}$
  - Compute *precise* result for that path
  - Merge ($\sqcap$) with all other precise results

$$\mathbf{out}_{b_i} = \bigsqcap_{[p_0,\ldots,p_k] \in path_{b_i}} trans_{b_i} \circ trans_{p_k} \circ \cdots \circ trans_{p_0}(\top)$$

Notation: (*function composition*)

$$(f \circ g)(x) = f(g(x))$$

MOP vs MFP: Example



Transfer functions

$$
\begin{aligned}
trans_{b_0} &= id \\
trans_{b_1} &= [x \mapsto 3][y \mapsto 1] \\
trans_{b_2} &= [x \mapsto 1][y \mapsto 3] \\
trans_{b_3} &= [z \mapsto x + y]
\end{aligned}
$$

Paths

$$
\begin{aligned}
path_{b_0} &= \{[]\} \\
path_{b_1} &= \{[b_0]\} \\
path_{b_2} &= \{[b_0]\} \\
path_{b_3} &= \{[b_0, b_1], [b_0, b_2]\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{out}_{b_3} &= ([z \mapsto x + y][x \mapsto 3][y \mapsto 1](\top)) \sqcap ([z \mapsto x + y][x \mapsto 1][y \mapsto 3](\top)) \\
&= \{z \mapsto 3 + 1, x \mapsto 3, y \mapsto 1\} \sqcap \{z \mapsto 1 + 3, x \mapsto 1, y \mapsto 3\} \\
&= \{z \mapsto 4, x \mapsto \bot, y \mapsto \bot\}
\end{aligned}
$$

MOP vs MFP

	MOP	MFP
Soundness	sound	sound
Precision	*maximal*	*sometimes lower*
Decidability	*undecidable*	*decidable*

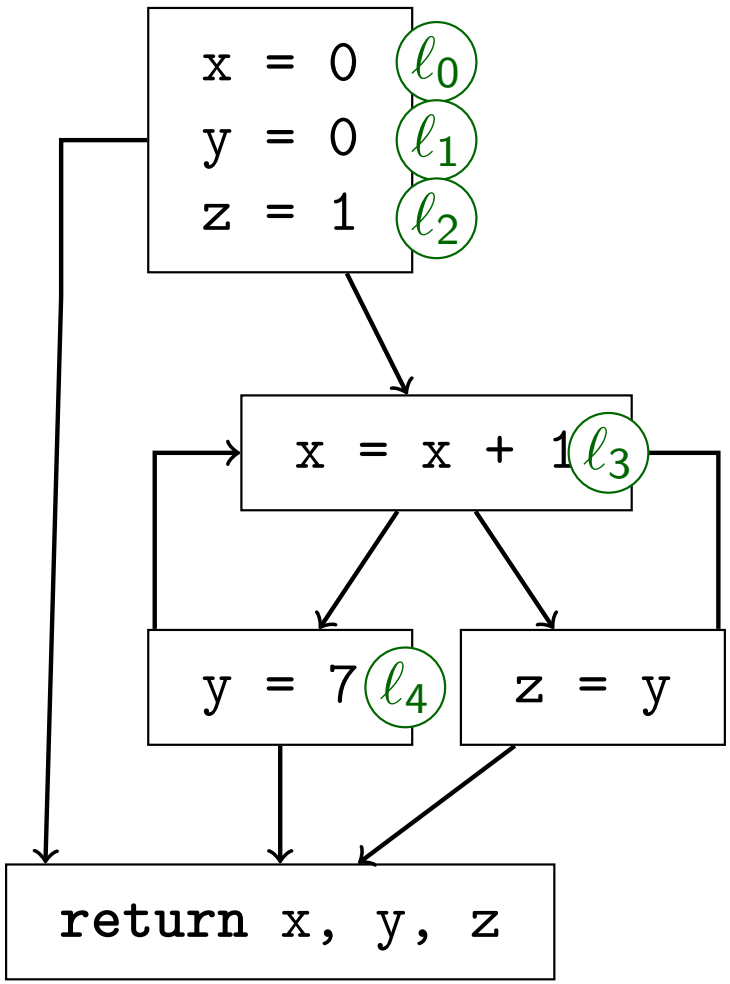
- MOP: Meet Over all Paths
- MFP: Maximal Fixed Point

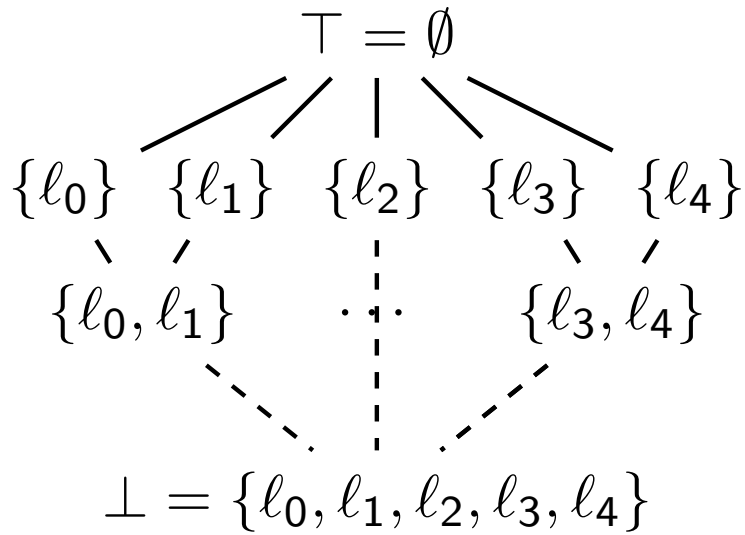
Summary

- $path_b$: Set of all paths from program start to b
- MOP: alternative to MFP (theoretically)
  - Termination not guaranteed
  - May be more precise
  - Idea:
    - Enumerate all paths to basic block
    - Compute transfer functions over paths individually
    - Meet

MFP revisited

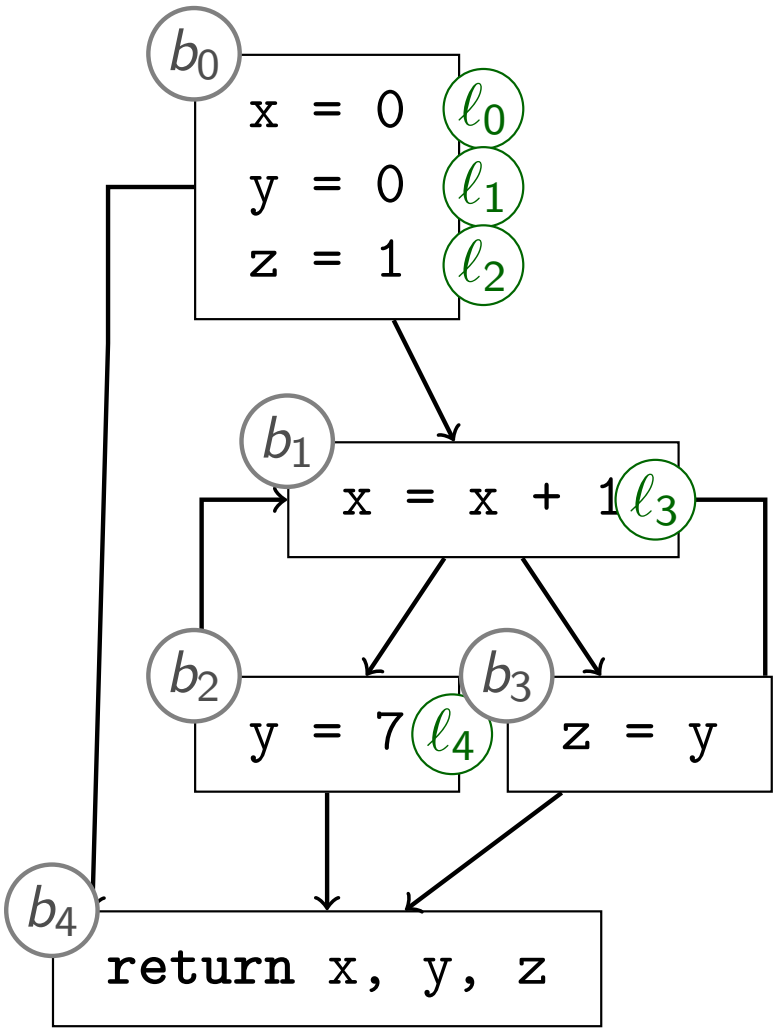
Consider **Reaching Definitions** again, with different lattice:





- All subsets of $\{\ell_0, \ldots, \ell_4\}$
- Finite height
- $\sqcap = \cup$

MFP revisited: Transfer Functions



$$
\begin{aligned}
trans_{b_0} = [\ & x \mapsto \{\ell_0\}, \\
& y \mapsto \{\ell_1\}, \\
& z \mapsto \{\ell_2\}] \\
trans_{b_1} = [\ & x \mapsto \{\ell_3\}] \\
trans_{b_2} = [\ & y \mapsto \{\ell_4\}] \\
trans_{b_3} = [\ & z \mapsto y]
\end{aligned}
$$

MOP vs MFP revisited

Solutions for b_4:

MOP solution

$$\begin{array}{lcl} x & \mapsto & \{\ell_0, \ell_3\} \\ y & \mapsto & \{\ell_1, \ell_4\} \\ z & \mapsto & \{\ell_1, \ell_2, \ell_4\} \end{array}$$

MFP solution

$$\begin{array}{lcl} x & \mapsto & \{\ell_0, \ell_3\} \\ y & \mapsto & \{\ell_1, \ell_4\} \\ z & \mapsto & \{\ell_1, \ell_2, \ell_4\} \end{array}$$

- Repeat with other programs:
  - MOP solution *always* the same as MFP solution
- Not true for other lattices/transfer functions...

Distributive Frameworks

A *Monotone Framework* is:

- Lattice $L = \langle \mathcal{L}, \sqsubseteq, \sqcap, \sqcup \rangle$
- L has finite height (Descending Chain Condition)
- All $trans_b$ are monotonic
- Guarantees that MFP *conservatively approximates* MOP

A *Distributive Framework* is:

- A Monotone Framework, where additionally:
- $trans_b$ distributes over $\sqcap$:

$$trans_b(x \sqcap y) = trans_b(x) \sqcap trans_b(y)$$

  for all programs and all x, y, b
- Guarantees that MFP *is equal to* MOP

Distributive Problems

- Monotonic:

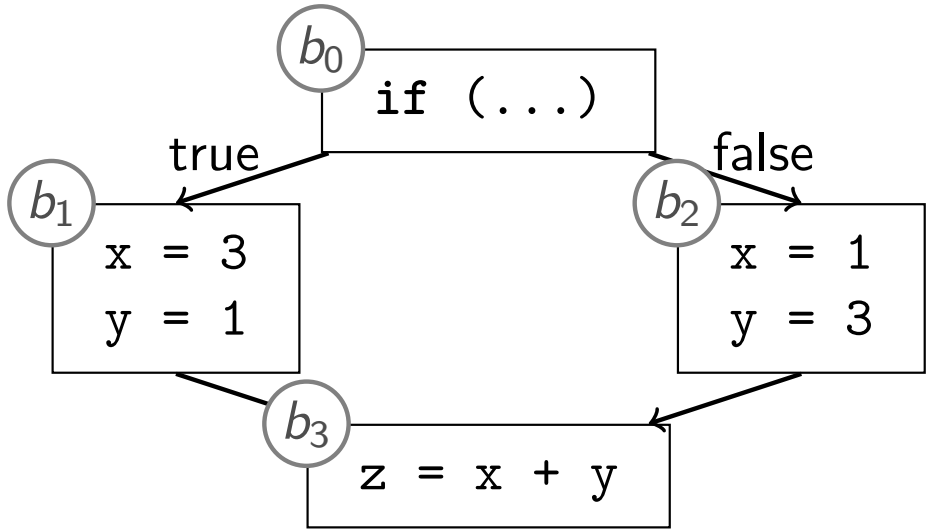
$$trans_b(x \sqcap y) \sqsubseteq trans_b(x) \sqcap trans_b(y)$$

- Distributive:

$$trans_b(x \sqcap y) = trans_b(x) \sqcap trans_b(y)$$

  - Many analyses can fit distributive framework
  - Known *counter-example*: transfer functions on $\mathbb{Z}^{\top}_{\bot}$:
    - $[z \mapsto x + y]$
    - Generally: transfer function that depends on two independent inputs and may produce same output for different inputs

A Hack to Improve Precision[1] (1/2)



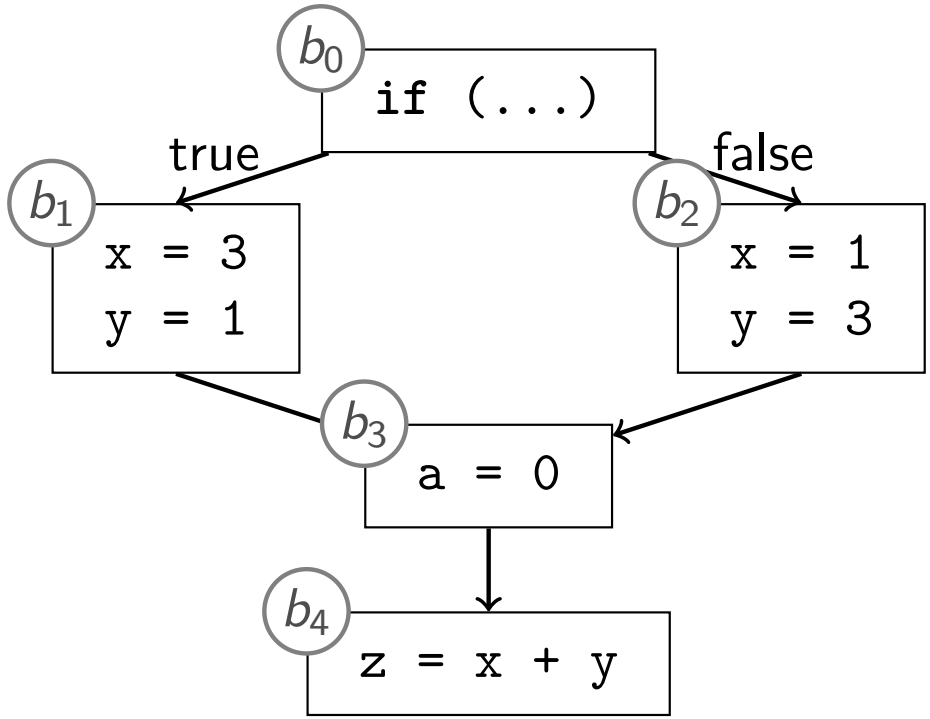
- Recall: Imprecision comes about because

$$\begin{aligned} trans_{b_3}(\mathbf{out}_{b_1} \sqcap \mathbf{out}_{b_2}) &= \\ trans_{b_3}(\{x \mapsto 3, y \mapsto 1, \ldots\} \sqcap \{x \mapsto 1, y \mapsto 3, \ldots\}) &= \\ trans_{b_3}(\{x \mapsto \top, y \mapsto \top, \ldots\}) &= \\ \{z \mapsto \top, \ldots\} & \end{aligned}$$

- *Idea*: Transfer first, *then* meet:

$$\begin{aligned} trans_{b_3}(\{x \mapsto 3, y \mapsto 1, \ldots\}) \sqcap trans_{b_3}(\{x \mapsto 1, y \mapsto 3, \ldots\}) &= \\ \{z \mapsto 4, \ldots\} \sqcap \{z \mapsto 4, \ldots\} & \end{aligned}$$

A Hack to Improve Precision (2/2)



$$\mathbf{in}_{b_4} = \mathbf{out}_{b_3} =$$
$$trans_{b_3}(\{x \mapsto 3, y \mapsto 1, \ldots\}) \sqcap trans_{b_3}(\{x \mapsto 1, y \mapsto 3, \ldots\}) =$$
$$\{x \mapsto 3, y \mapsto 1, \ldots\} \sqcap \{x \mapsto 1, y \mapsto 3, \ldots\} =$$
$$\{x \mapsto \top, y \mapsto \top, \ldots\}$$

Only works if data is used right at point of the merge

Summary

- *Distributive Frameworks* are *Monotone Frameworks* with additional property:

$$trans_b(x \sqcap y) = trans_b(x) \sqcap trans_b(y)$$

  for all programs and all x, y, b
- In Distributive Frameworks, MOP and MFP produce same answer
- Gen/Kill-set based analyses are always distributive

Subroutine calls

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & | & \mathit{id}\ .\ \mathit{id} \\
 & & \\
\mathit{val} & ::= & \langle \mathit{name} \rangle \\
 & | & \mathit{num} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{val} \rangle + \langle \mathit{val} \rangle \\
 & | & \text{null} \\
 & | & \text{print } \langle \mathit{val} \rangle \\
 & | & \text{new()} \\
 & | & \mathit{id}\ (\ \langle \mathit{val} \rangle \star\ )
\end{array}
\qquad
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle = \langle \mathit{expr} \rangle \\
 & | & \{\ \langle \mathit{stmt} \rangle \star\ \} \\
 & | & \text{if } \langle \mathit{val} \rangle\ \langle \mathit{stmt} \rangle \text{ else } \langle \mathit{stmt} \rangle \\
 & | & \text{while } \langle \mathit{val} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \text{skip} \\
 & | & \text{return } \langle \mathit{val} \rangle \\
 & & \\
\mathit{decl} & ::= & \text{proc } \mathit{id}\ (\ \mathit{id} \star\ )\ \langle \mathit{stmt} \rangle \\
 & | & \text{global } \mathit{id} = \mathit{num} \\
 & & \\
\mathit{prog} & ::= & \mathit{decl} \star
\end{array}
$$

Limitations of Intra-Procedural Analysis

ATL

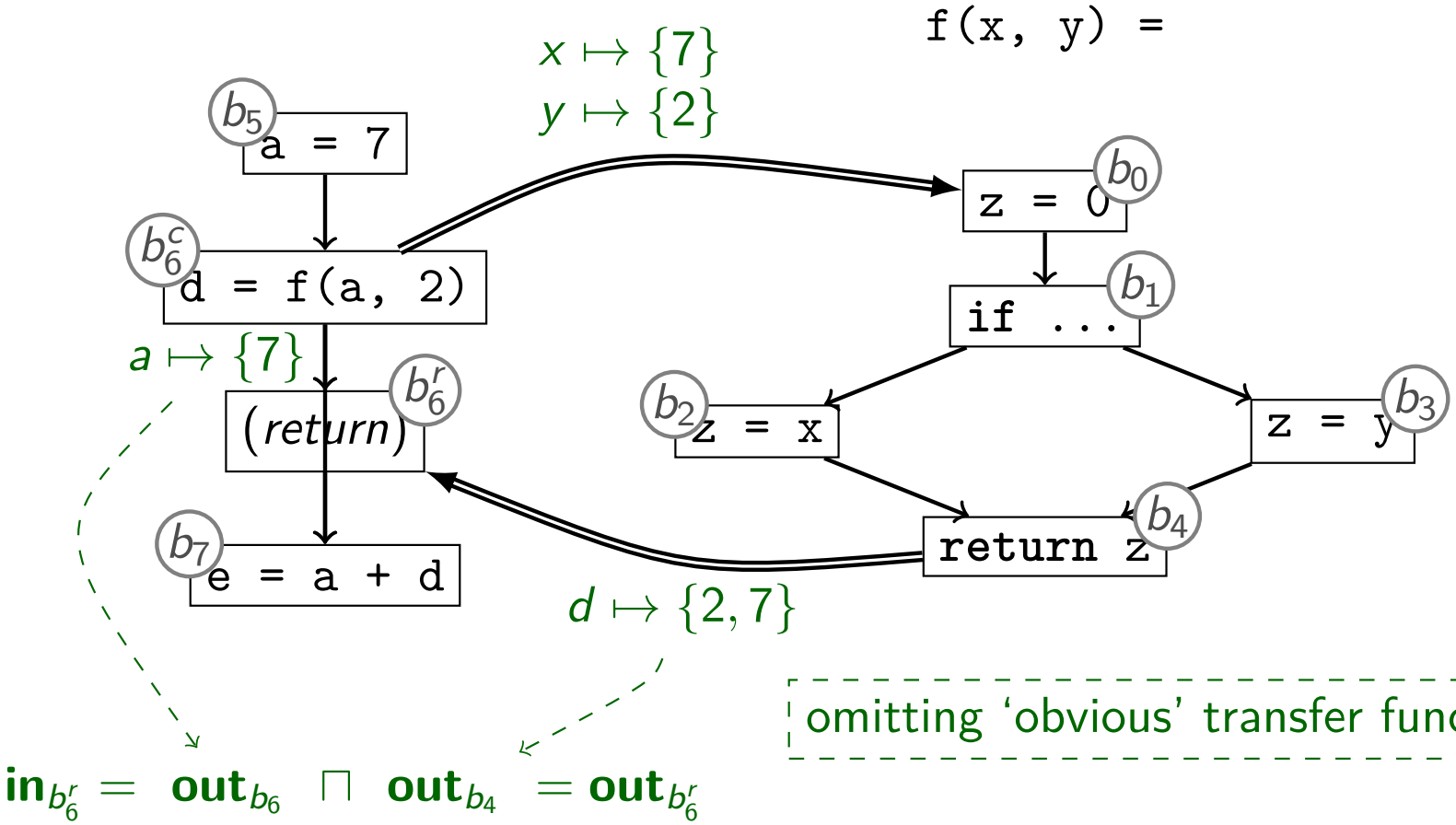
```
a = 7
d = f(a, 2)
e = a + d
```

ATL

```
proc f(x, y) {
  z = 0
  if x > y {
    z = x
  } else {
    z = y
  }
  return z
}
```

How can we compute Reachable Definitions here?

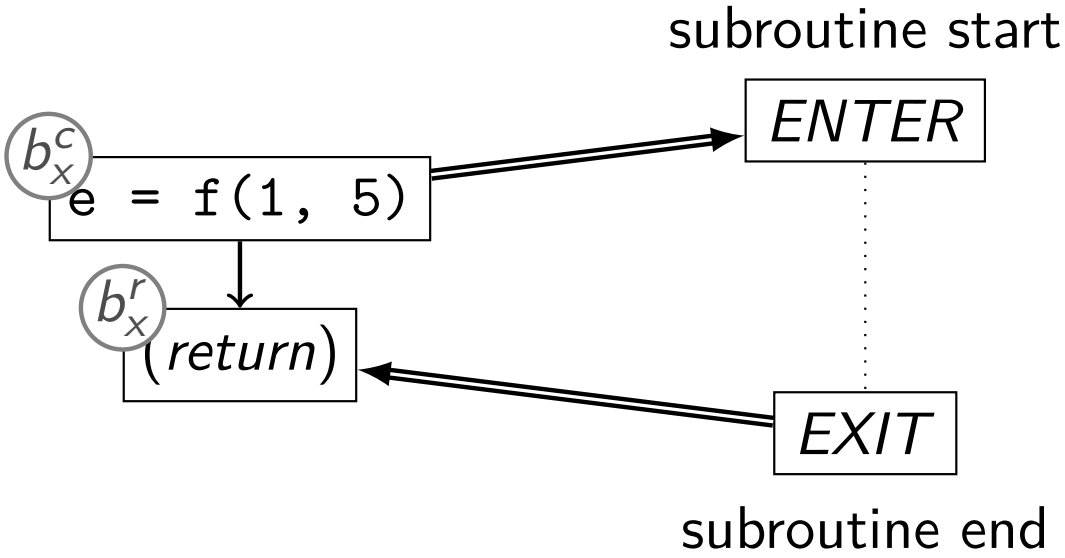
A Naive Inter-Procedural Analysis



- $\mathbf{out}_{b_7}$: $e \mapsto \{9, 14\}$

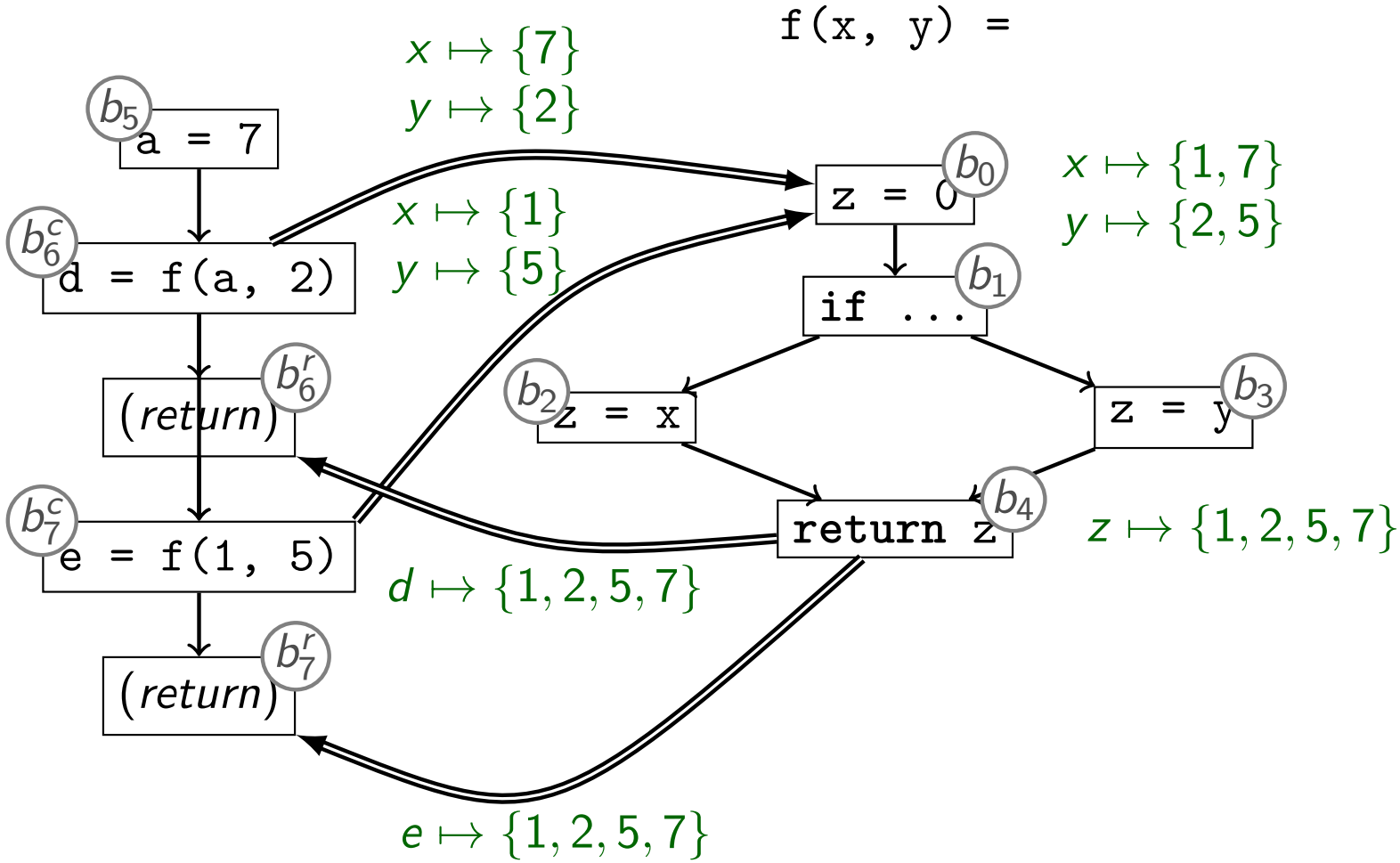
Works rather straightforwardly!

Inter-Procedural Data Flow Analysis



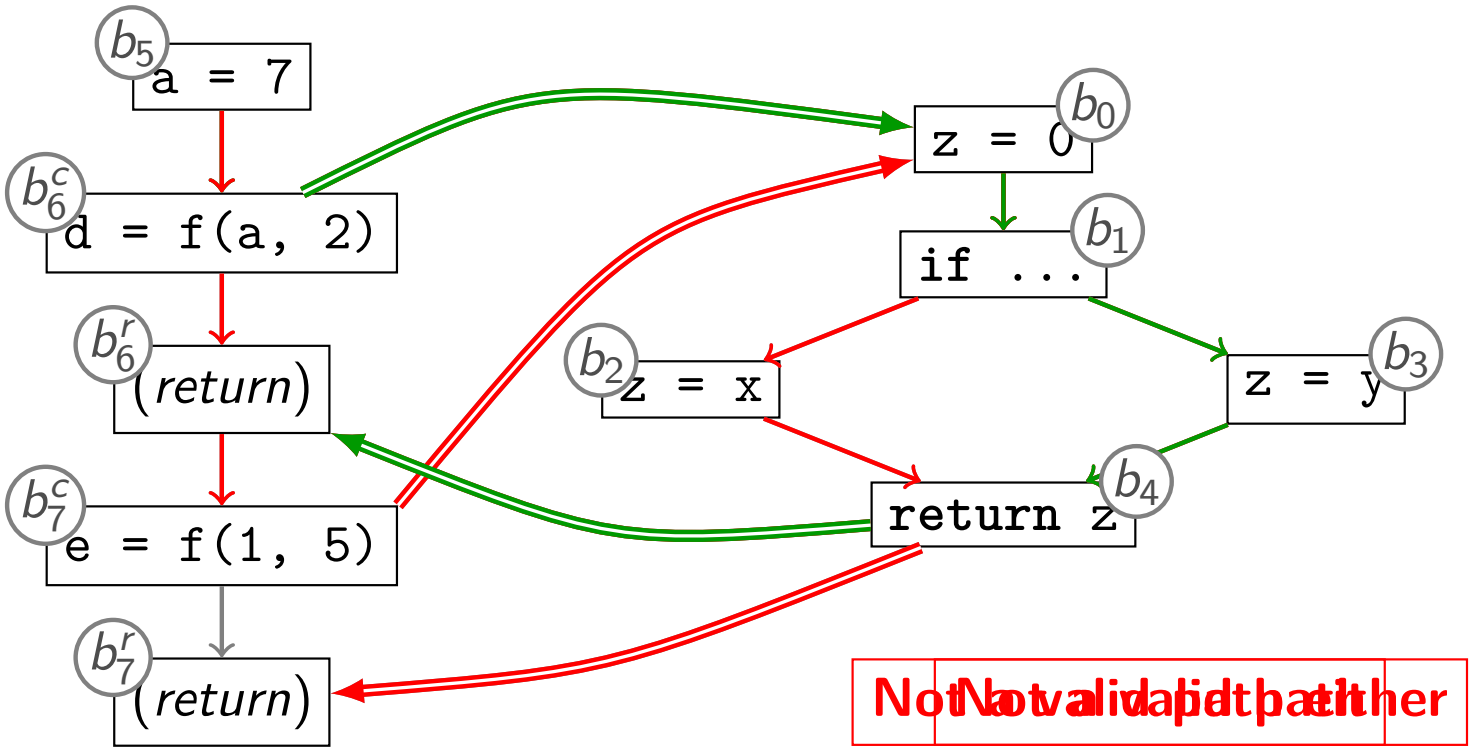
- Split call sites b_x into *call* (b_x^c) and *return* (b_x^r) nodes
- Intra-procedural edge $b_x^c \longrightarrow b_x^r$ carries environment/store
- Inter-procedural edge ($\Longrightarrow$):
  - Caller $\Longrightarrow$ subroutine, substitutes parameters (for pass-by-value)
  - Caller $\Longleftarrow$ return, substitutes result (for pass-by-result)
  - Otherwise as intra-procedural data flow edge

A Naive Inter-Procedural Analysis



Imprecision!

Context Sensitivity: Valid Paths



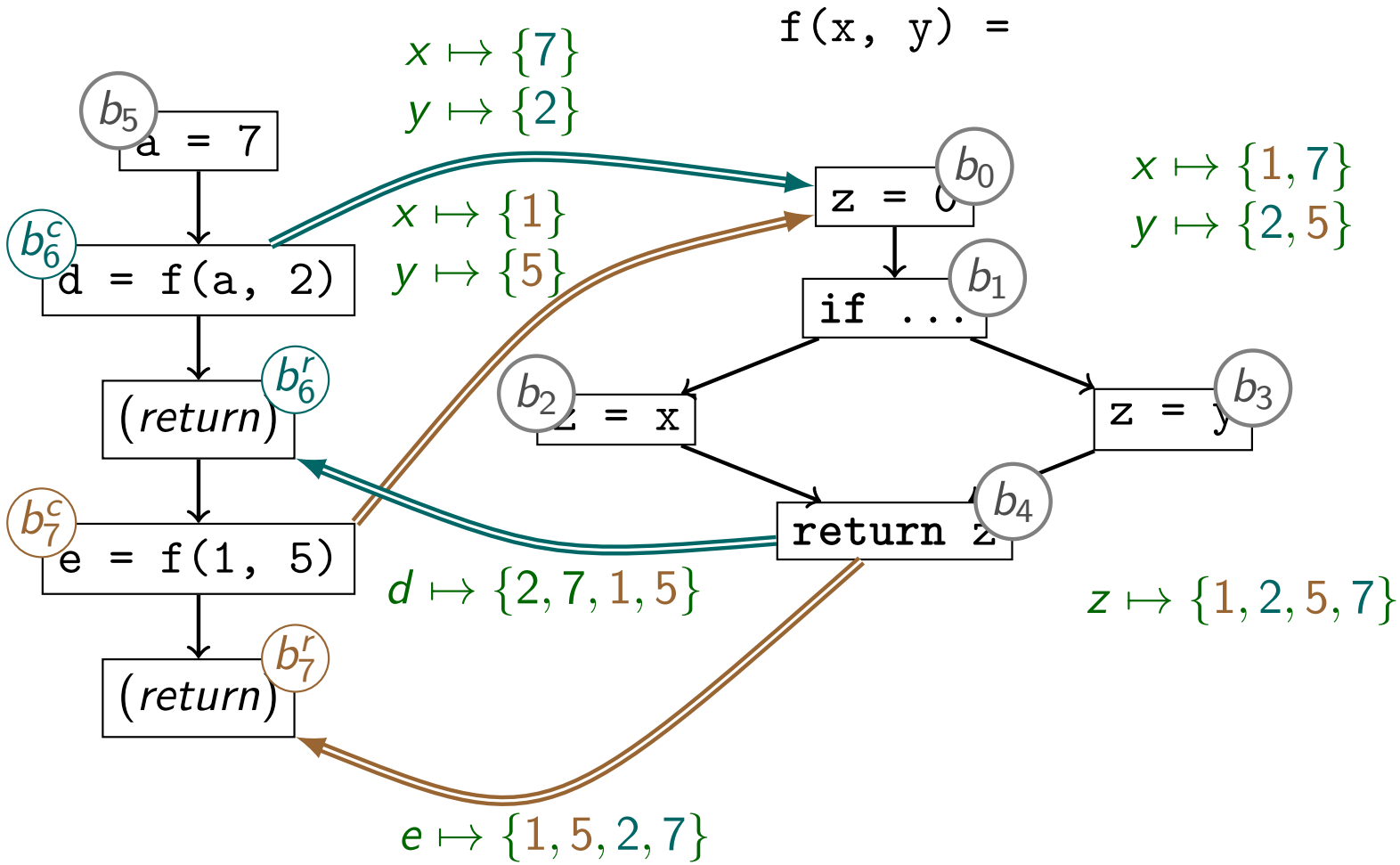
- $[b_5, b_6^c, b_0, b_1, b_3, b_4, b_6^r]$

Context-sensitive analyses consider only valid paths

Summary

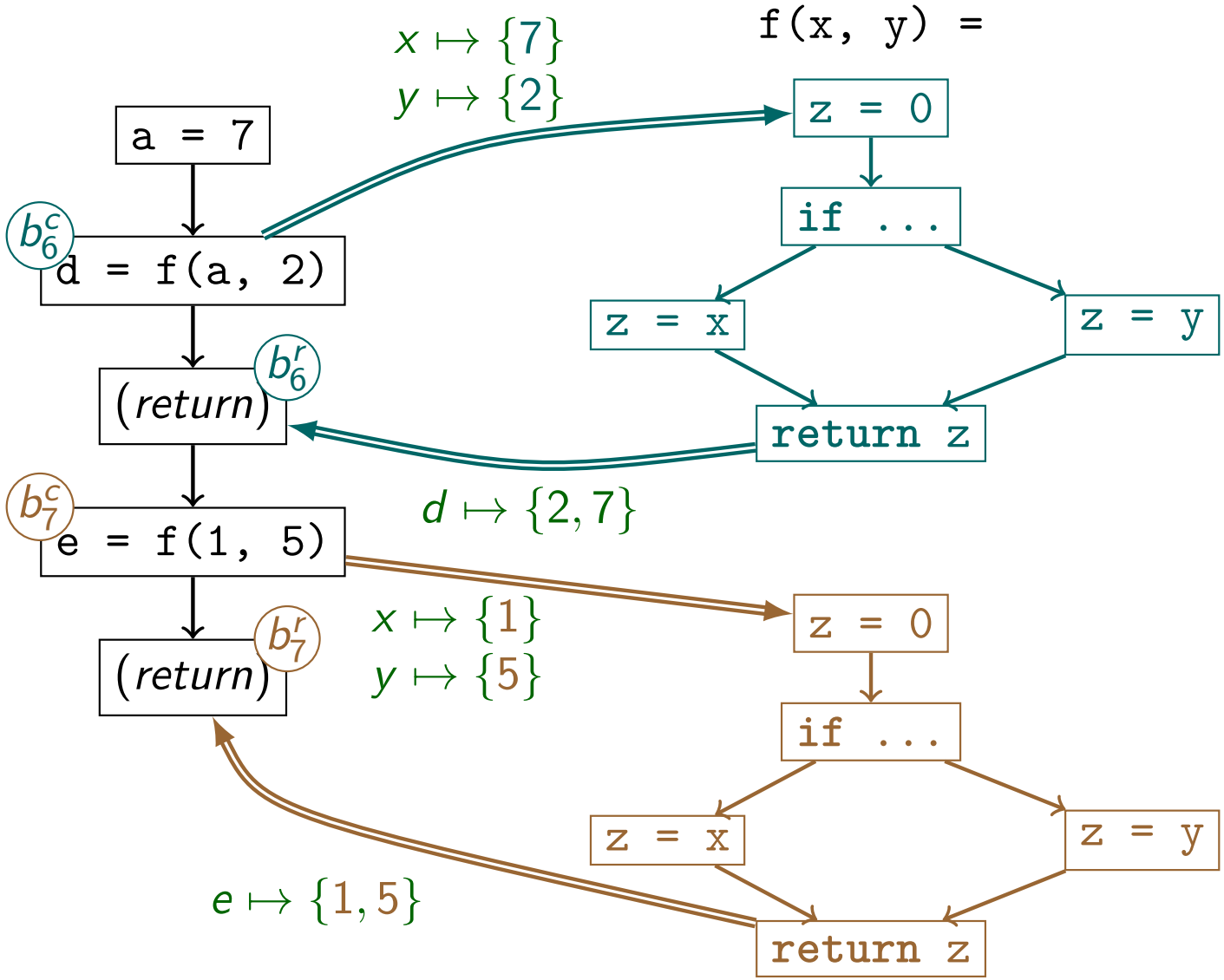
- **Intraprocedural** Data Flow Analysis is highly imprecise with subroutine calls
- **Interprocedural** Data Flow Analysis is more precise:
  - Split call site into call site + return site
  - Add flow edges between call sites, subroutine entry
  - Add flow edges between subroutine return, return site
  - Carry environment from call site to return site
- Interprocedural analysis must typically consider the entire program
  $\Rightarrow$ **whole-program analysis**
- Naive interprocedural analysis is **context-insensitive**
  - Merge all callers into one

Interprocedural Data Flow Analysis



***Context-insensitive*: analysis merges all callers to `f()`**

Inlining



Clone subroutine IRs for each *calling context*

Summary

- **Context-sensitive** analysis distinguishes 'calling context' when analysing subroutine
  - 'Who called me'?
  - Can go deeper: 'And who called them?'
- **Inlining** is one strategy for **context-sensitive** analysis
- Copy subroutine bodies for each caller
- **Advantages:**
  - Simple
  - Improves precision
- **Disadvantages:**
  - Difficult with recursion
  - Slows down analysis

Review

- Gen/Kill analyses
- Lattices
- Monotone Frameworks
- MFP algorithm
- MOP algorithm
- Distributive Frameworks
- Interprocedural Analysis
  - Inlining for analysis

To be continued...

Next week:

- More on IFDS and its refinements
- Callgraph Analysis