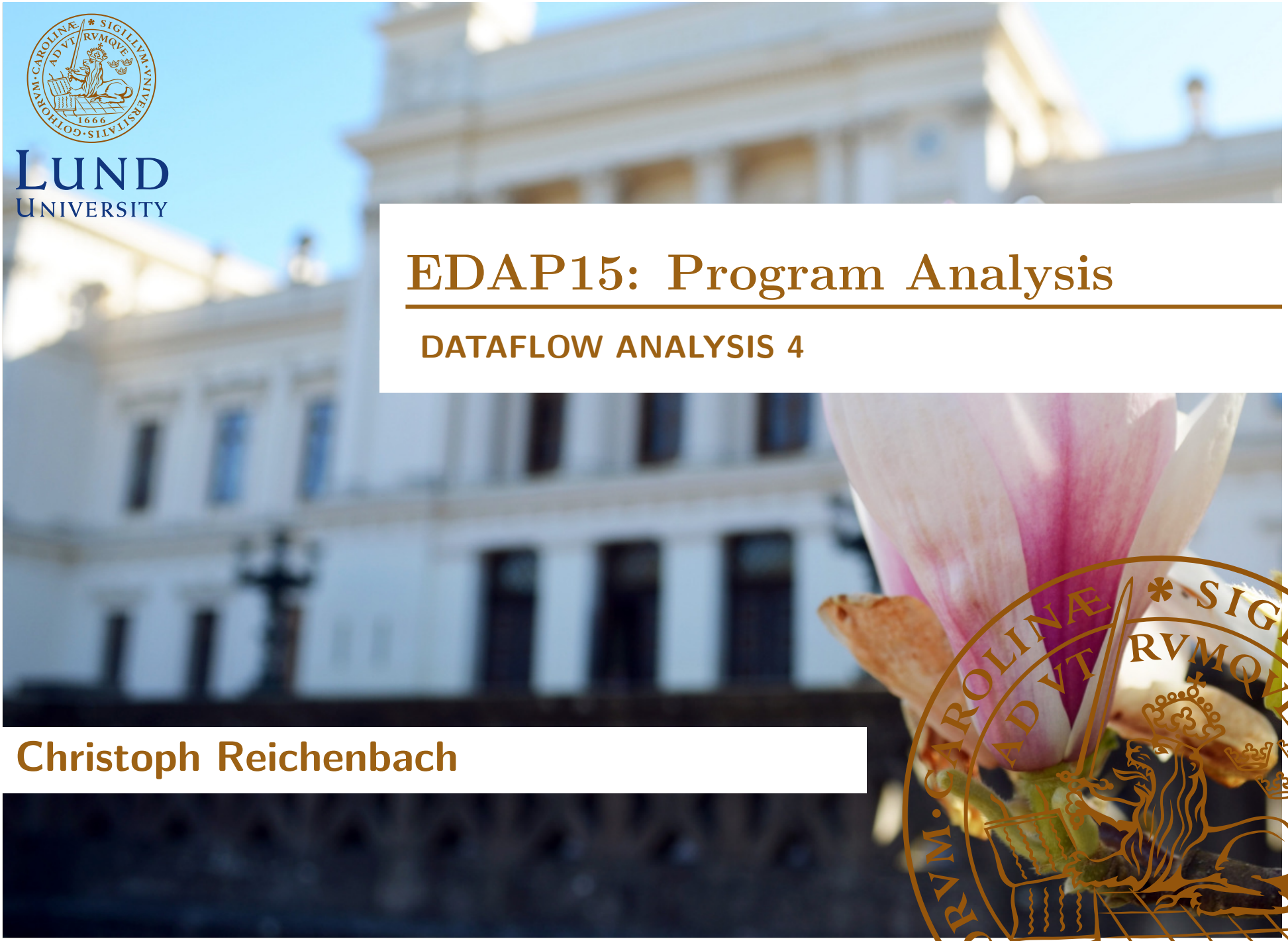


LUND UNIVERSITY

EDAP15: Program Analysis

DATAFLOW ANALYSIS 4

Christoph Reichenbach

Welcome back!

- Lab 3 is up
  - Will release update for code-prober soon
    - Solves potentially misleading error message affecting lab 3b
- **Guest Lecture**
  - Patrik Åberg, Magnus Templing from Ericsson
  - Lecture 13, 3rd of March
  - Background for Lab 4 (final lab)
- **Lab 2 & 3 solo lab slots**
- **Presenting lab 2 or 3 to me**
  - During lab, otherwise I will e-mail you

Questions?

Today's Topics

- **Loops**

```
var x := 16;
while read() != "" {
    x := x / 2;
}
var y := 20 - x;      // [[y]] = ?
```

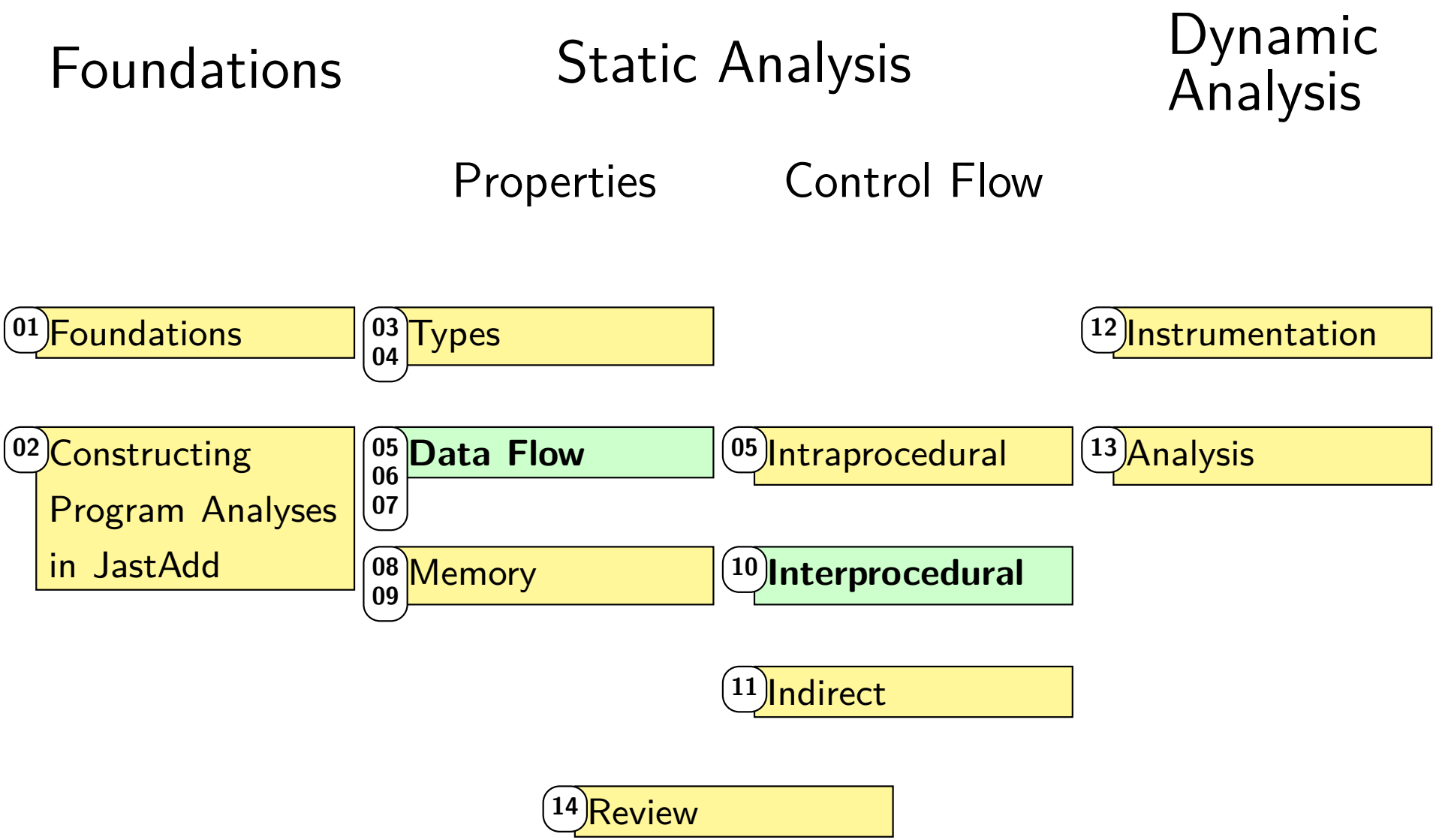
- **Conditionals**

```
a[0] := 1;        Warning:  a = null possible
if a != null {
    return a[0]; // No warning here?
}
```

- **Function Calls**

```
fun f() = return null;
fun g() = {
  var a := f();
  return a[0];    Warning:  a = null possible
}
```

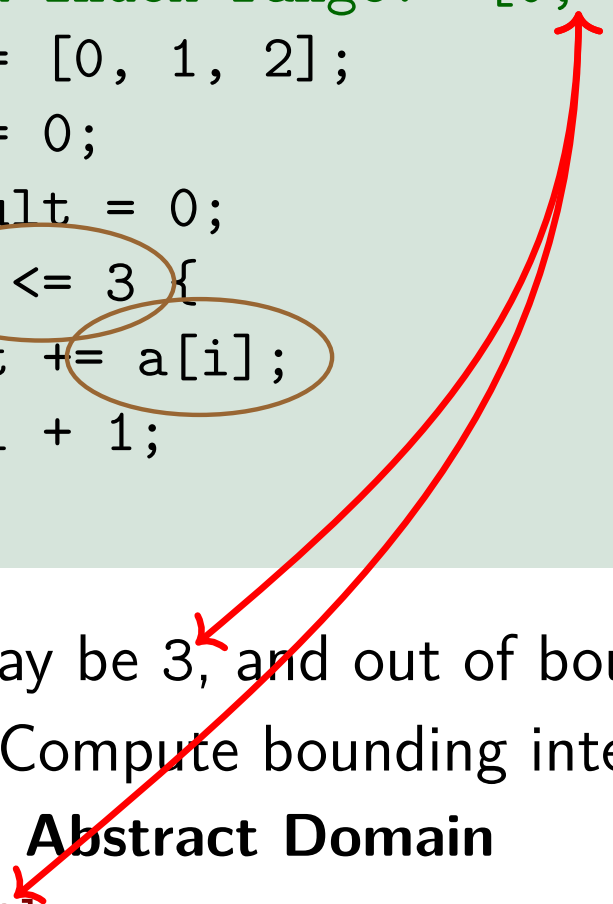
Lecture Overview



The Interval Domain: Motivation

Teal

```
// valid index range:  [0, 2]
var a := [0, 1, 2];
var i := 0;
var result = 0;
while i <= 3 {
  result += a[i];
  i := i + 1;
}
```



- Bug: `i` may be 3, and out of bounds for a
- Analysis: Compute bounding intervals $[min, max]$
  - **Interval Abstract Domain**
- $[\![\texttt{i}]\!] = [0, 3]$

Interval Domain: Examples

Abstract values: $[l, r]$ where $l \leq r$, plus $\bot$

Teal

```
var x := 5;
var y := random(1, 10);
var z := x + y;
var w := z - y;
```

$[\![\texttt{x}]\!] = [5, 5]$

$[\![\texttt{y}]\!] = [1, 10]$

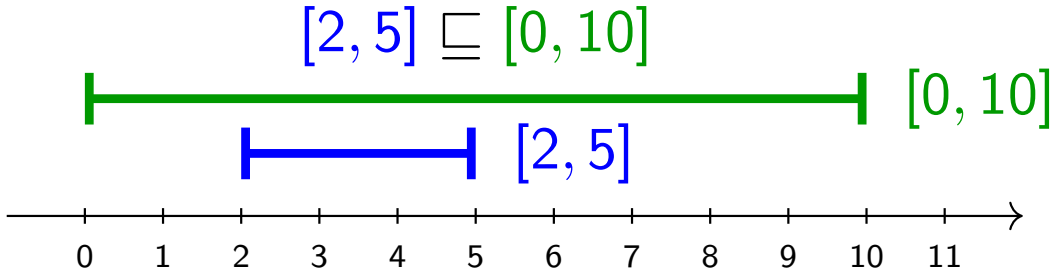
$[\![\texttt{z}]\!] = [6, 15]$

$[\![\texttt{w}]\!] = [-4, 14]$

- **Constants:** $[\![n]\!] = [n, n]$
- **Addition:** $[\![e_1 + e_2]\!] = [l_1 + l_2, r_1 + r_2]$ where $[\![e_1]\!] = [l_1, r_1]$ and $[\![e_2]\!] = [l_2, r_2]$
- **Subtraction:** $[\![e_1 - e_2]\!] = [l_1 - r_2, r_1 - l_2]$ where $[\![e_1]\!] = [l_1, r_1]$ and $[\![e_2]\!] = [l_2, r_2]$

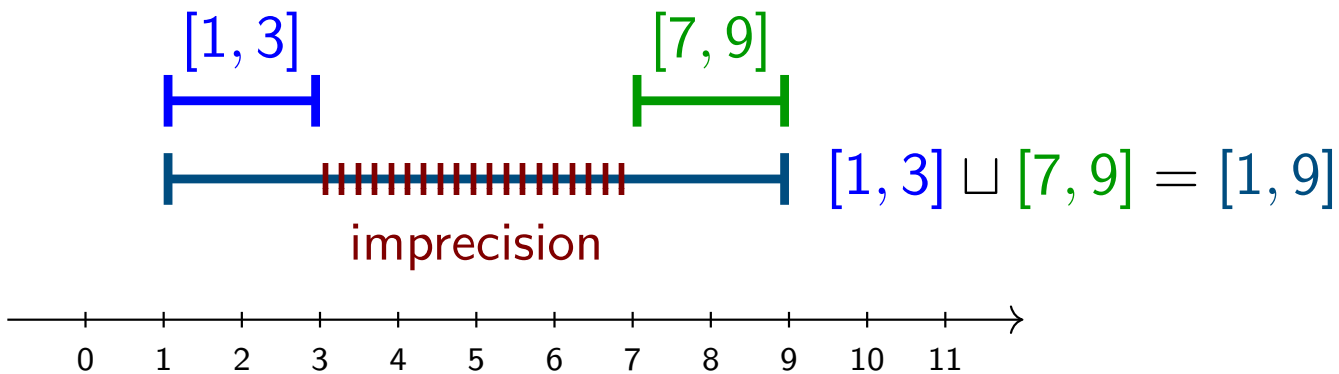
Interval Domain: Ordering and Join

Ordering: $[l_1, r_1] \sqsubseteq [l_2, r_2]$ iff $l_1 \geq l_2$ and $r_1 \leq r_2$



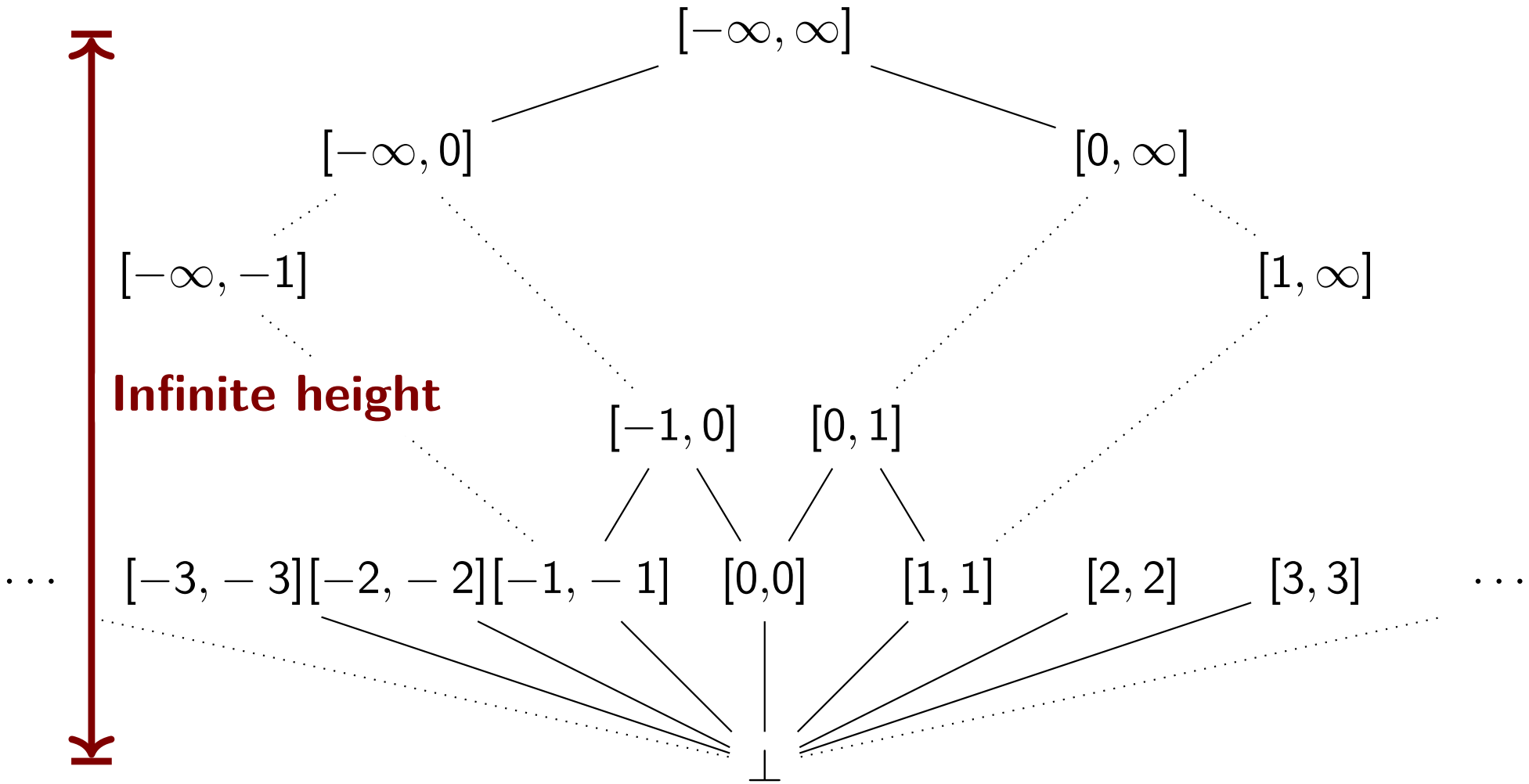
- Smaller intervals are more precise

Join: $[l_1, r_1] \sqcup [l_2, r_2] = [min(l_1, l_2), max(r_1, r_2)]$



- Smallest interval containing both inputs

Interval Domain Lattice



- $\top = [-\infty, \infty]$
- $\bot$: distinct value (empty interval)

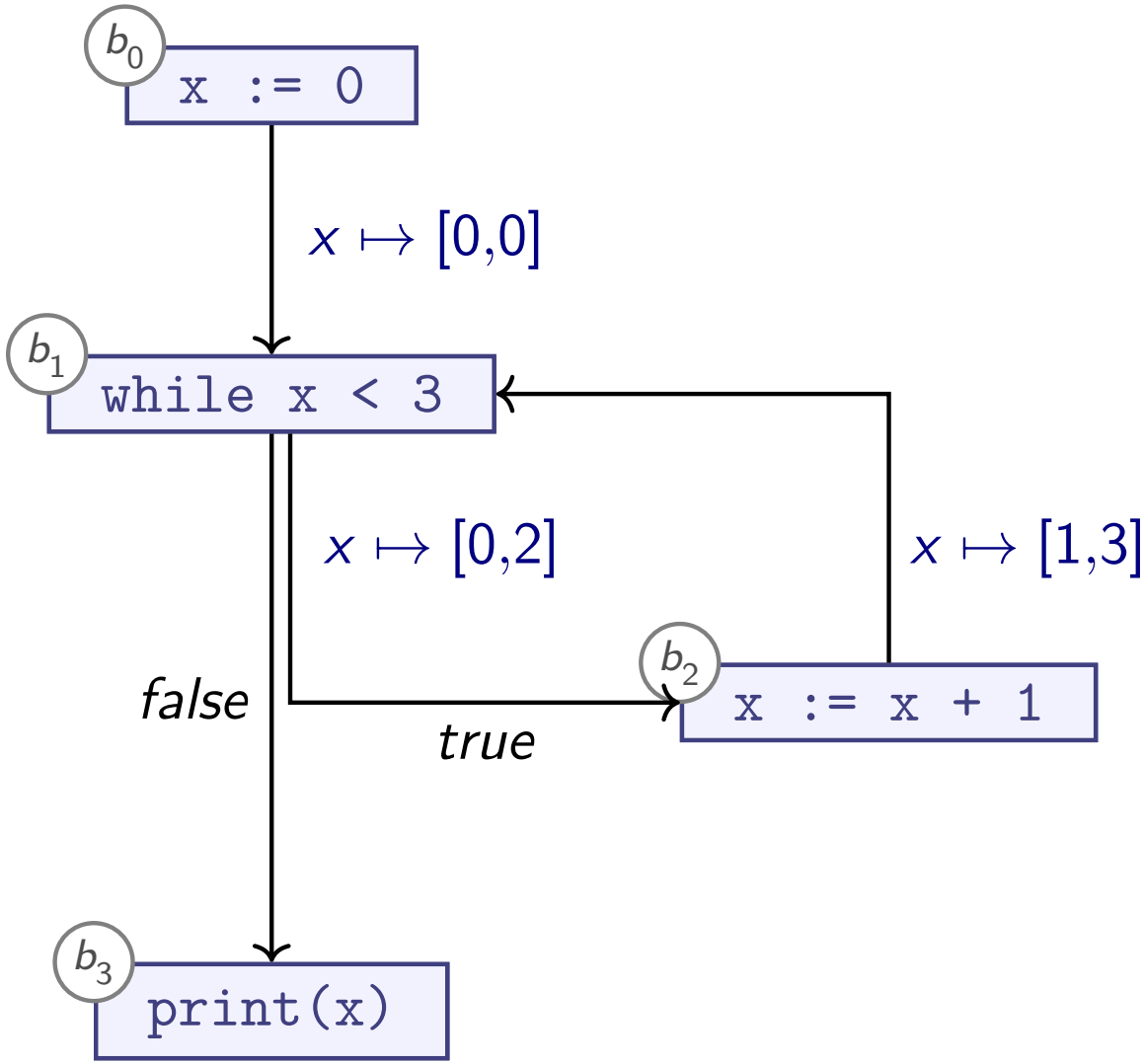
Summary: Interval Abstract Domain

- Approximate numbers as *intervals*:

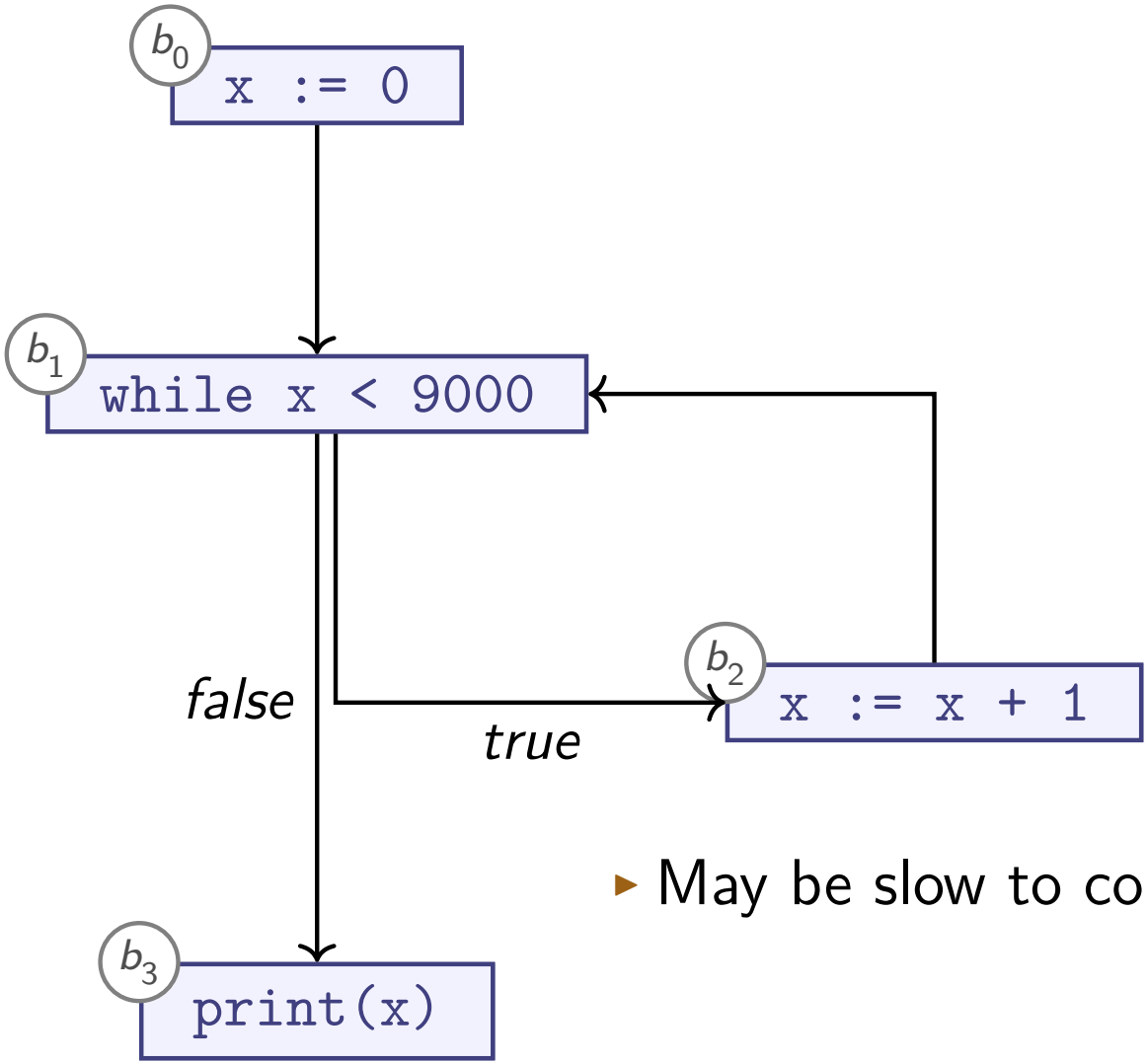
$$[l, r] \text{ where } l, r \in \mathbb{Z}, l \leq r$$

- $\top = [-\infty, \infty] = \mathbb{Z}$
- $\bot = \emptyset$ is the empty interval
- Infinite lattice height: No termination guarantee with our current tools

Applying the Interval Domain

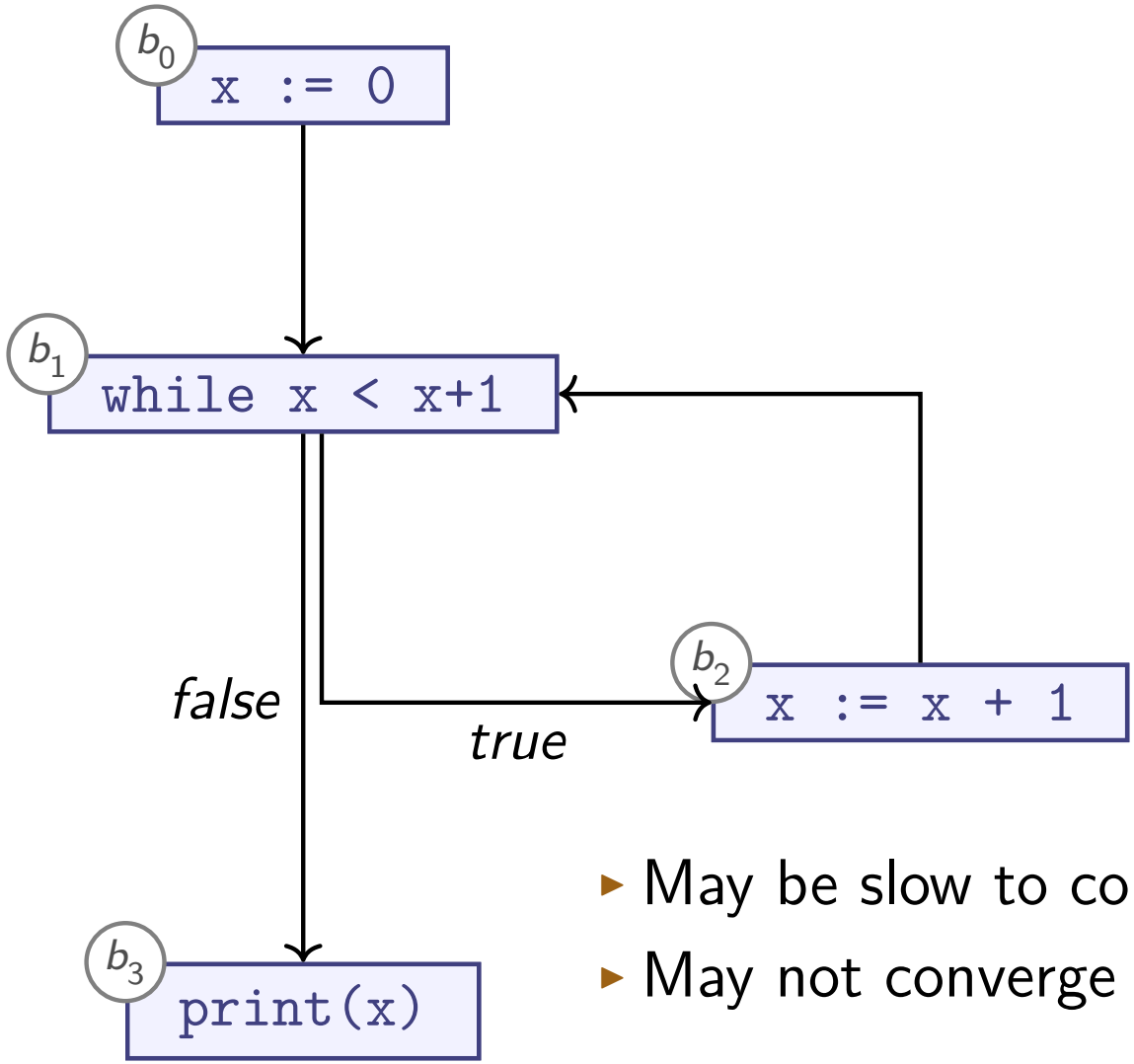


Applying the Interval Domain



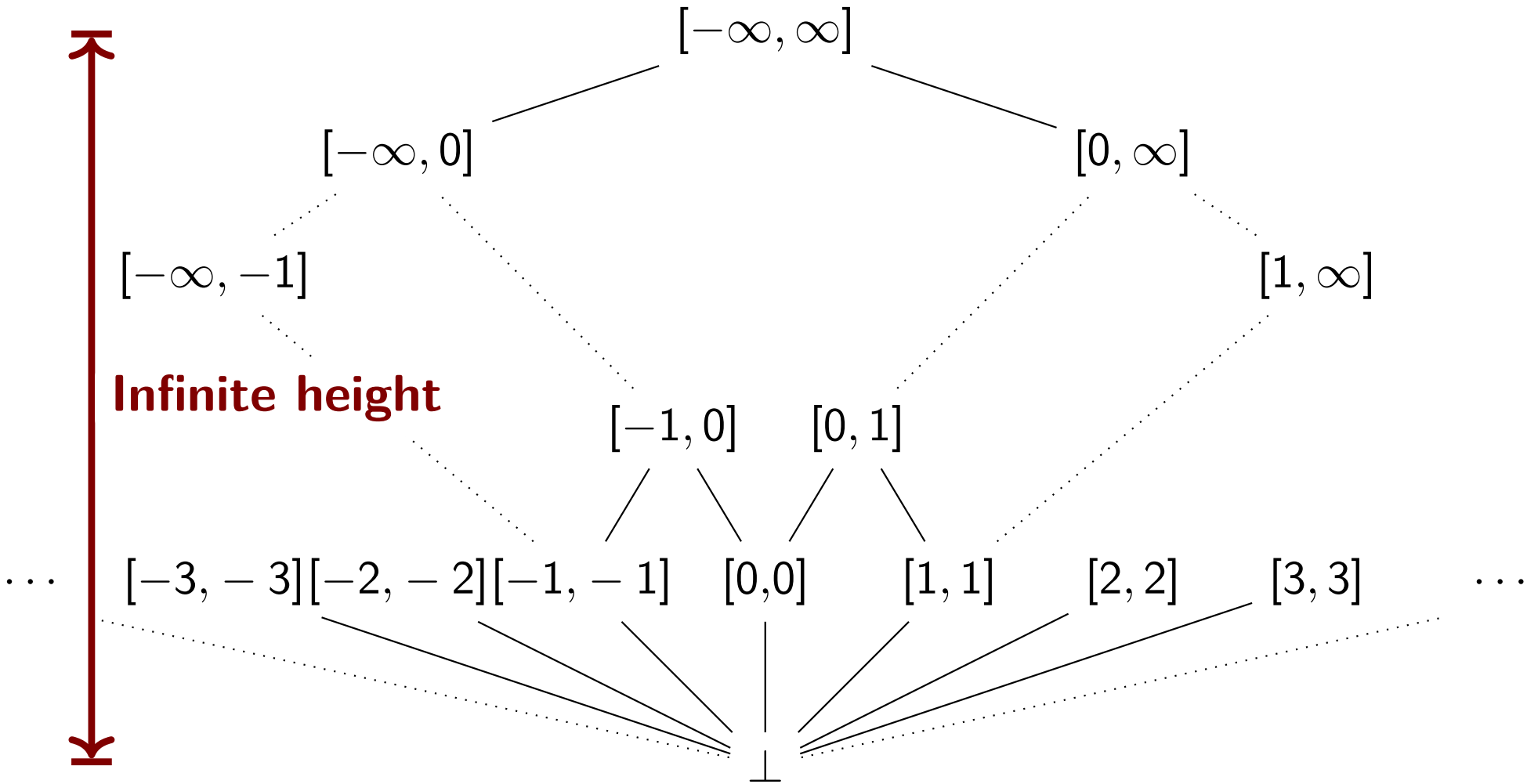
- May be slow to converge

Applying the Interval Domain

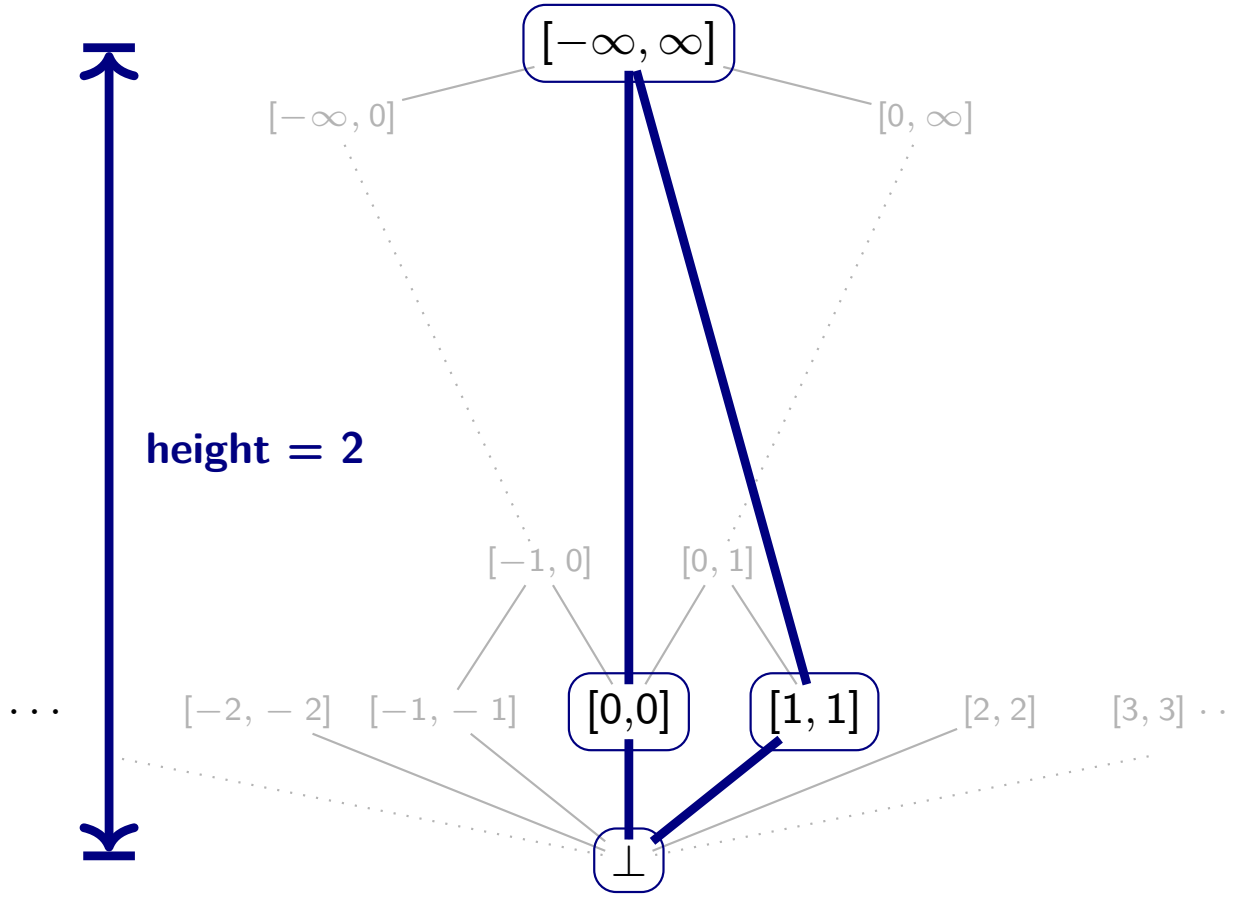


- May be slow to converge
- May not converge at all

Interval Domain Lattice



Finite Sublattices for Convergence

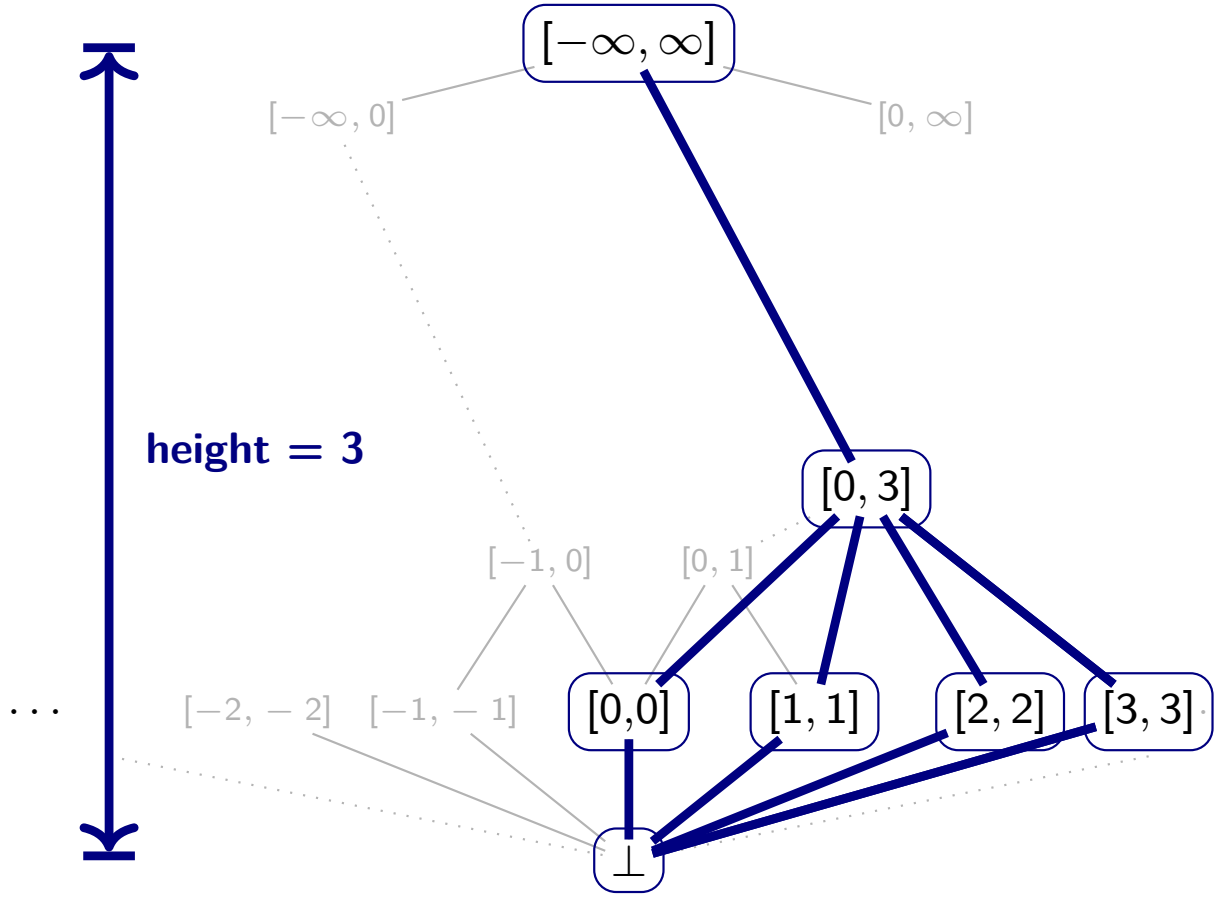



```
x := 0;
while x < 3 {
  x := x + 1;
}
```

```
x := 0;
while x < x+1 {
  x := x + 1;
}
```

```
x := 0;
while x < 9000 {
  x := x + 1;
}
```

Finite Sublattices for Convergence

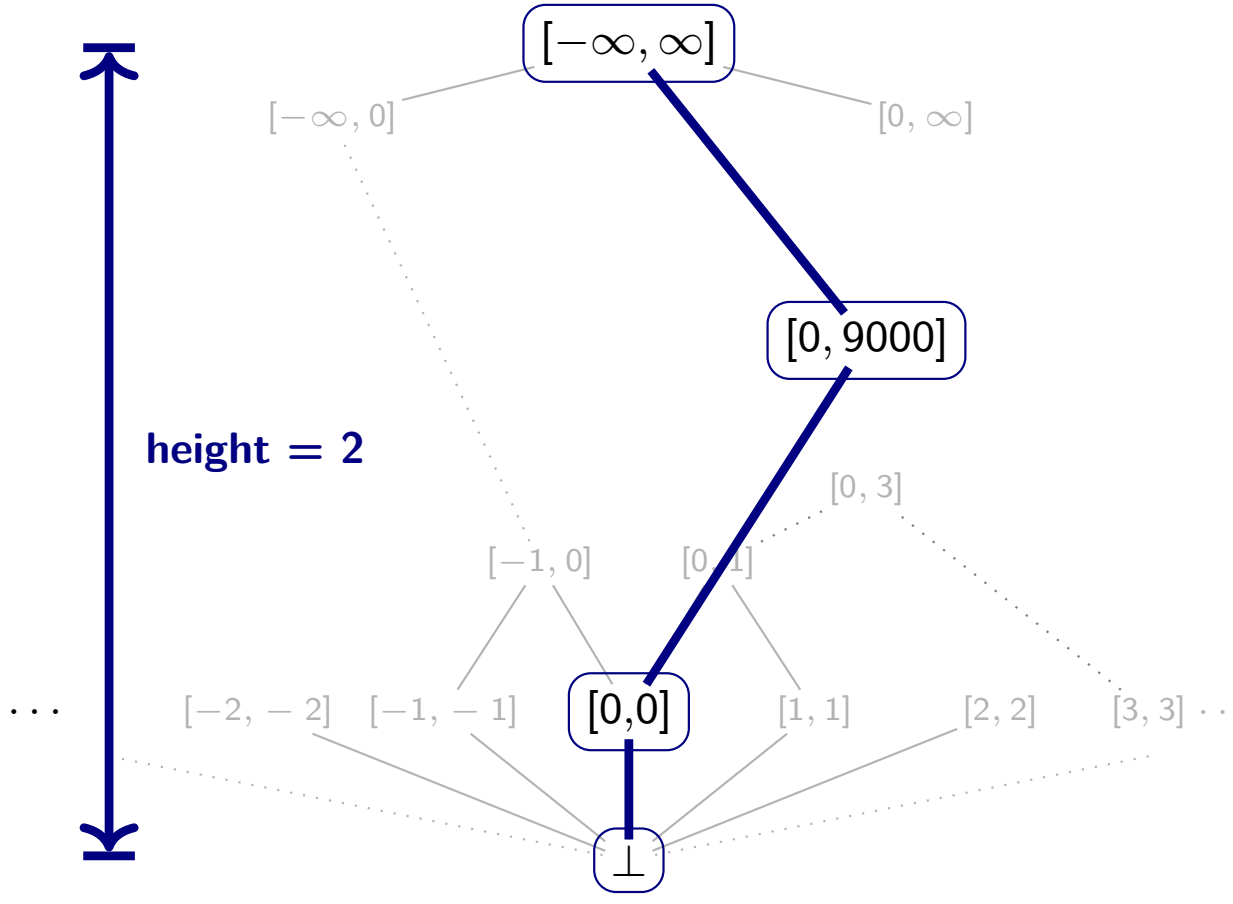



```
x := 0;
while x < 3 {
  x := x + 1;
}
```

```
x := 0;
while x < x+1 {
  x := x + 1;
}
```

```
x := 0;
while x < 9000 {
  x := x + 1;
}
```

Finite Sublattices for Convergence




```
x := 0;
while x < 3 {
  x := x + 1;
}
```

```
x := 0;
while x < x+1 {
  x := x + 1;
}
```

```
x := 0;
while x < 9000 {
  x := x + 1;
}
```

Sublattices of the Interval Lattice

- For convergence: extract **Finite Sublattice**
- **Challenges:**
  - Which sublattice?
  - Best choice is **program-dependent**
  - Ideally: different lattices at different points in CFG
- **On-demand** lattice selection: use *widening*:

$$v_1 \nabla v_2 = \begin{cases} v_1 & \iff \quad v_1 = v_2 \\ \textbf{widen}(v_1 \sqcup v_2) & \iff \quad v_1 \neq v_2 \end{cases}$$

- For a suitable **widen** function

Widening

- For convergence: satisfy Ascending Chain Condition on:

$$v_{i+1} = \textbf{widen}(v_i)$$

- Suitable functions for Interval Domain?
  - $\textbf{widen}_{\top}(v) = \top$
    - Very conservative
    - Ensures convergence
  - $\textbf{widen}_{10000}([l,r]) = [l - 10000, r + 10000]$
    - *No convergence*: still allows infinite ascending chain
  - $\textbf{widen}_{\mathcal{K}}([l,r]) = [max(\{v \in \mathcal{K} | v < l\}), min(\{v \in \mathcal{K} | v > r\})]$
    - Ensures convergence *iff* $\mathcal{K}$ *is finite*
    - Must pick "good" $\mathcal{K}$
    - Common strategy:
      $\mathcal{K} = \{-\infty, \infty\} \cup$ all numeric literals in program
      Our example: $\mathcal{K} = \{-\infty, 0, 1, 9000, \infty\}$

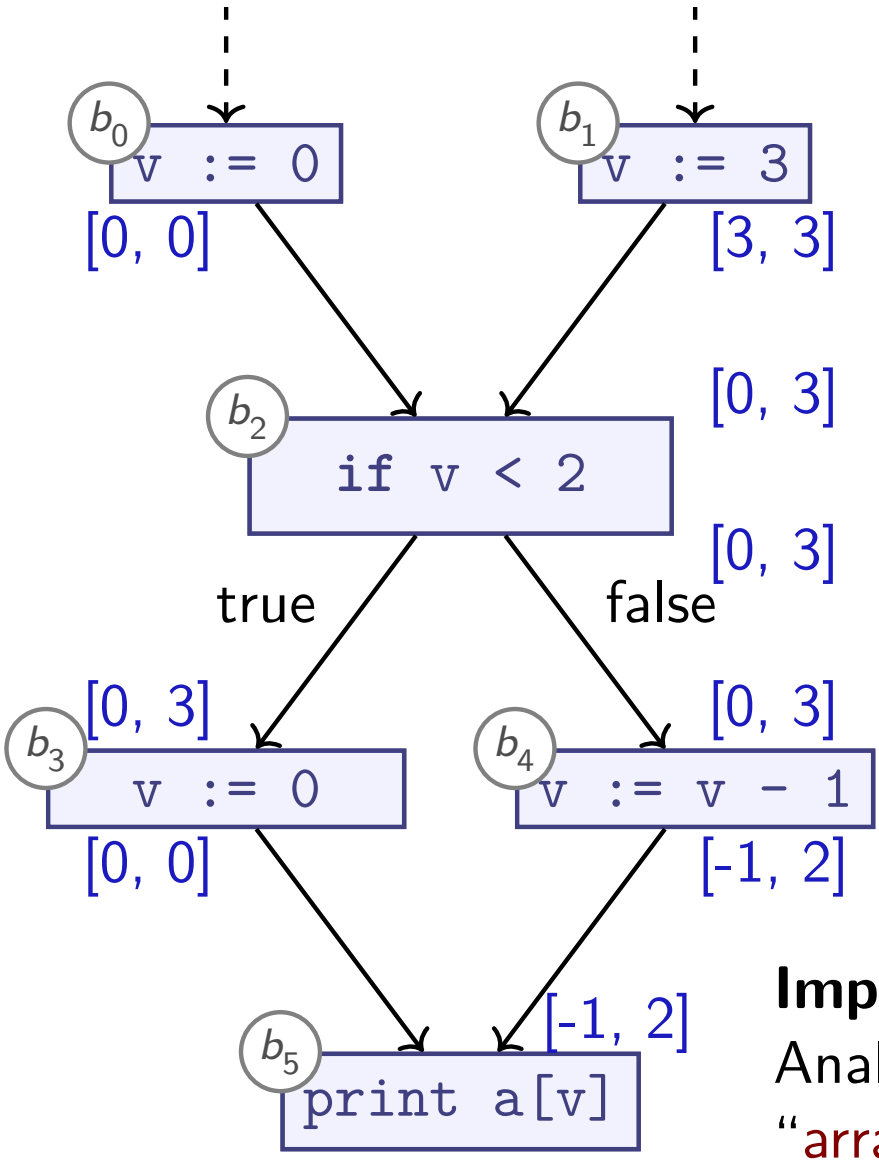
```
var x := 0;
while x < 9000 {
    x := x + 1;
}
```

Summary

- **Widening** allows us to use infinite domains $\mathcal{L}$
- Define **widen** function:
  - **widen** must satisfy Ascending Chain Condition on $\mathcal{L}$
  - **widen**($\mathcal{L}$) generates finite lattice
- Widening operator ∇ applies **widen** function iff needed
- Approach:

1. Before analysis runs: we design analysis on infinite-height lattice
2. When analysis runs on concrete program:
   - **widen** constructs finite-height lattice specific to program (or specific to *CFG node*, even)
   - ∇ applies **widen** on demand
     MFP: When updating: $\mathbf{in}_j := \mathbf{in}_j \nabla \mathbf{out}_j$
   - In JastAdd dataflow analysis a:
     during a`Out()` or a`In()`

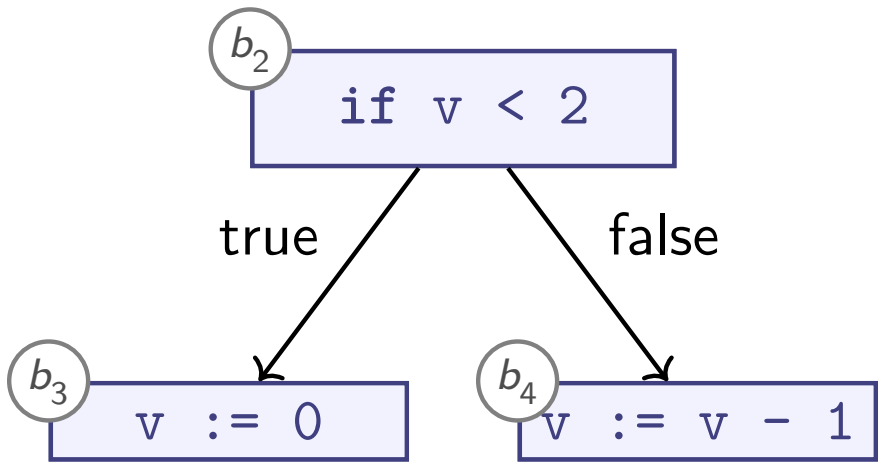
Conditionals



Imprecision can yield false positive
Analysis concludes:
"array index may be negative"

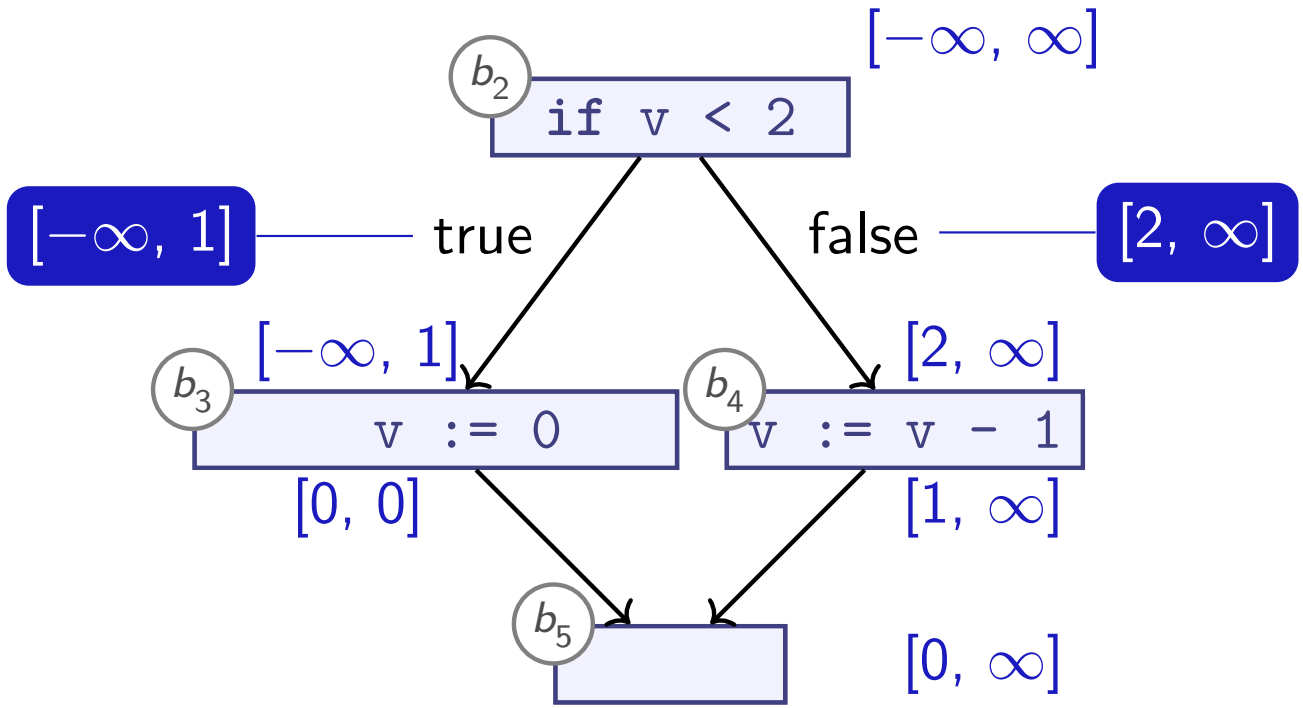
Handling Conditionals (1/2)

- So far: Did not make use of conditional predicate
  - `true` branch: only if `v < 2`
    $v \in [-\infty, 1]$
  - `false` branch: only if $\neg$`(v < 2)`
    $v \in [2, \infty]$



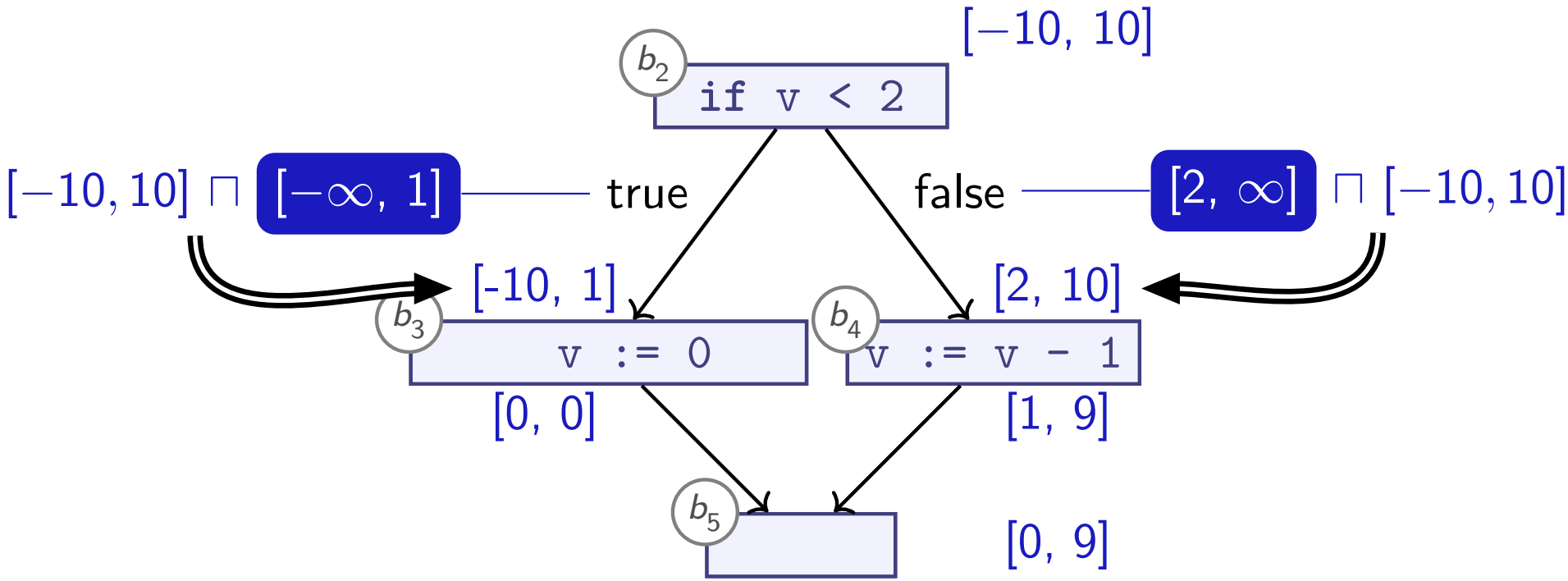
- **Control Sensitive** analysis utilises this information
  - Filter possible values

Handling Conditionals (2/2)



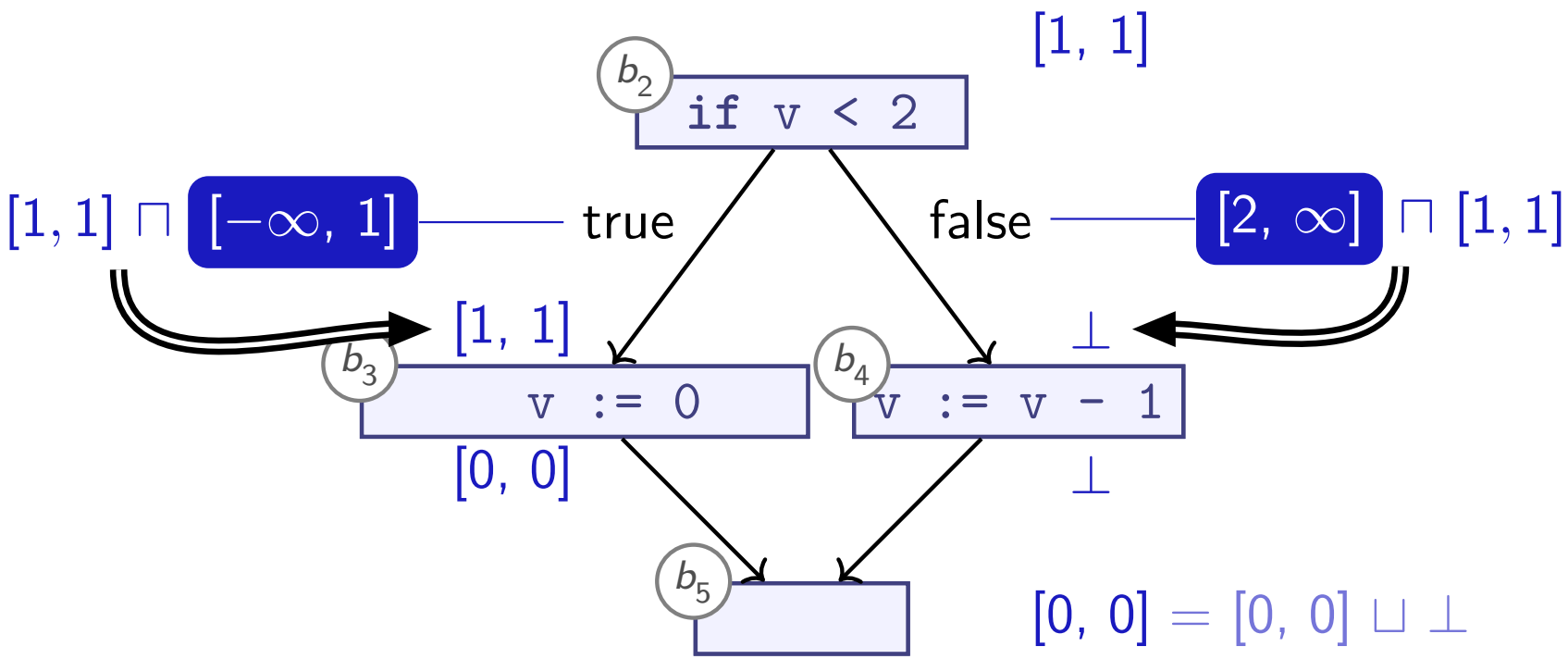
- *Idea*: Split interval for v for true/false branches
  - Analyse each branch with part of original interval
  - "Re-assemble" interval on join afterwards

Handling Conditionals (2/2)



- *Idea*: Split interval for v for true/false branches
  - Analyse each branch with part of original interval
  - "Re-assemble" interval on join afterwards
- If not $v \mapsto \top$:
  - **Filter** with lattice *meet*: $\sqcap$

Contradictions



Summary

- **Control sensitive** analysis considers conditionals:
  - May propagate different information along different edges:
    - **if** P:
    - Special transfer function for '**assert** P' on 'true' edge
    - Special transfer function for '**assert** `not` P' on 'false' edge
- More precise than **control insensitive** analysis
- Utilises *Lattice Meet* operation $\sqcap$

  Intuition: $a \sqcap b$ "satisfy a **and** b"
  $a \sqcup b$ "satisfy a **or** b"
- $a \sqcap b = \bot$ can happen: *branch will never execute*

Applications of Dataflow Analysis

- ✓ Value Domains: variables as abstract values

$$[\![\texttt{x * x}]\!] = [0, \infty]$$

  - Relational Domains: relations between variables:

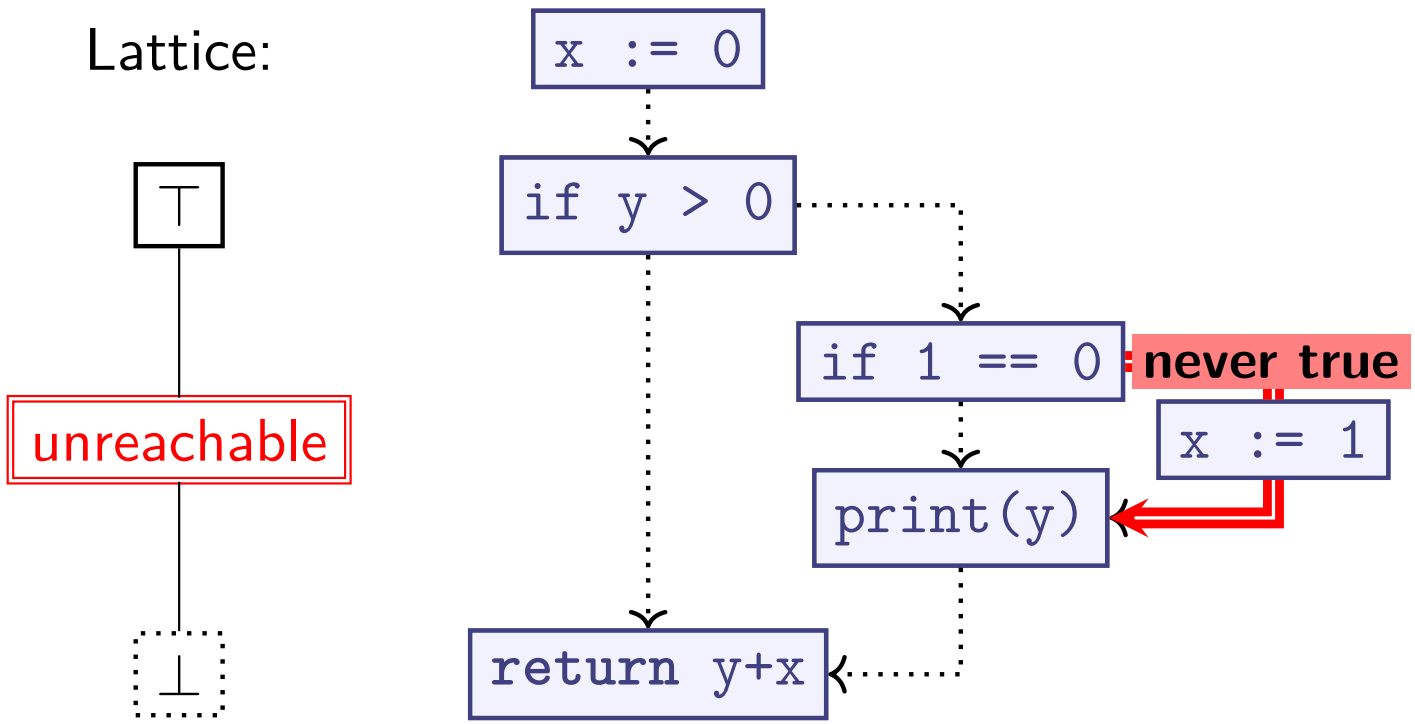
$$\texttt{x} < \texttt{y}$$

  - Control Domains:
    - Is code "live" / reachable?
  - Others:
    - Side effects
    - Unused Assignment
    - Available Expressions
  - Combinations of the above

Analysing Control Flor

Unreachable Path Elimination

- "Which CFG edges can never be taken?"
- Usually depends on constant propagation / folding
- *Forward* analysis



No need to distinguish between unreachable and ⊥

Summary

- Dataflow analysis can abstract over:
  - Values (per-variable)
  - Relations (between variables)
  - Control flow
  - Side effects

  ...

What about subroutines?

Teal

```
var x := pick_one(0, 5);
print(10 / x); // Division by zero?
```

- Understanding code usually requires understanding subroutines like `max`

Inter- vs. Intra-Procedural Analysis

- **Intra**procedural: Within one procedure
  - Data flow analysis so far
- **Inter**procedural: Across multiple procedures
  - Type Analysis, especially. with polymorphic type inference

Limitations of Intra-Procedural Analysis

Teal-0

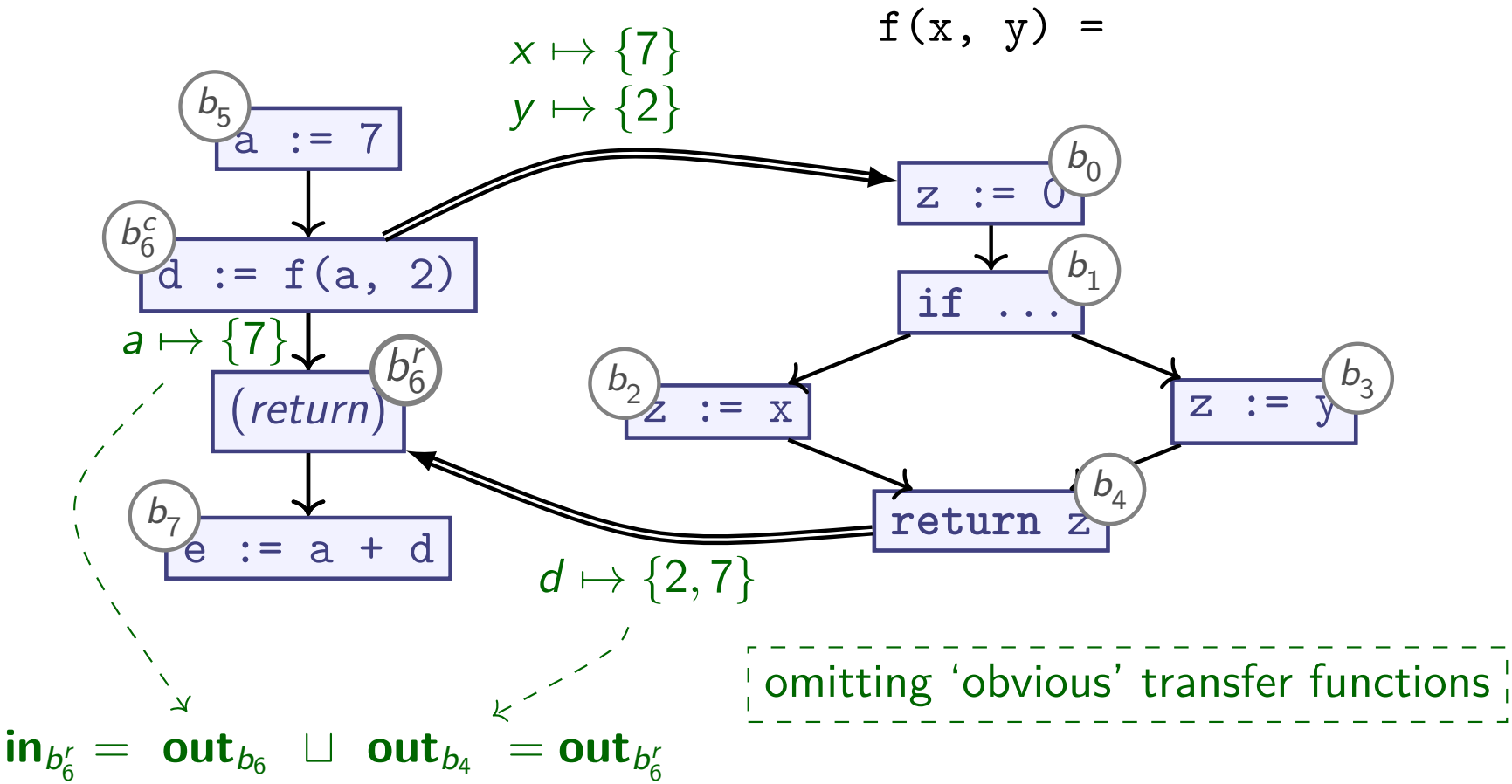
```
a := 7;
d := f(a, 2);
e := a + d;
```

Teal-0

```
fun f(x, y) = {
  var z := 0;
  if x > y {
    z := x;
  } else {
    z := y;
  }
  return z;
}
```

How can we compute Constant Propagation here?

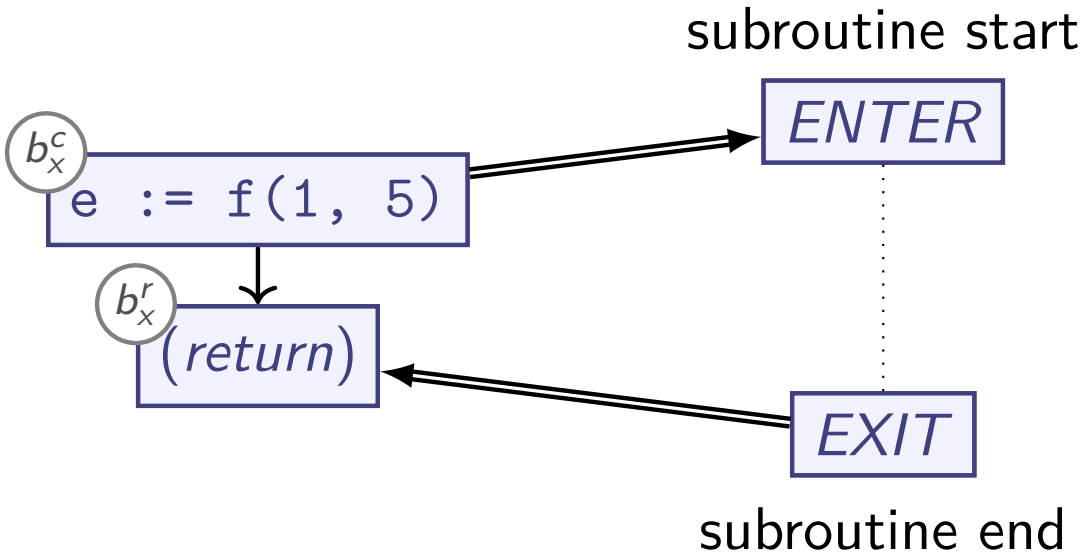
A Naïve Inter-Procedural Analysis



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

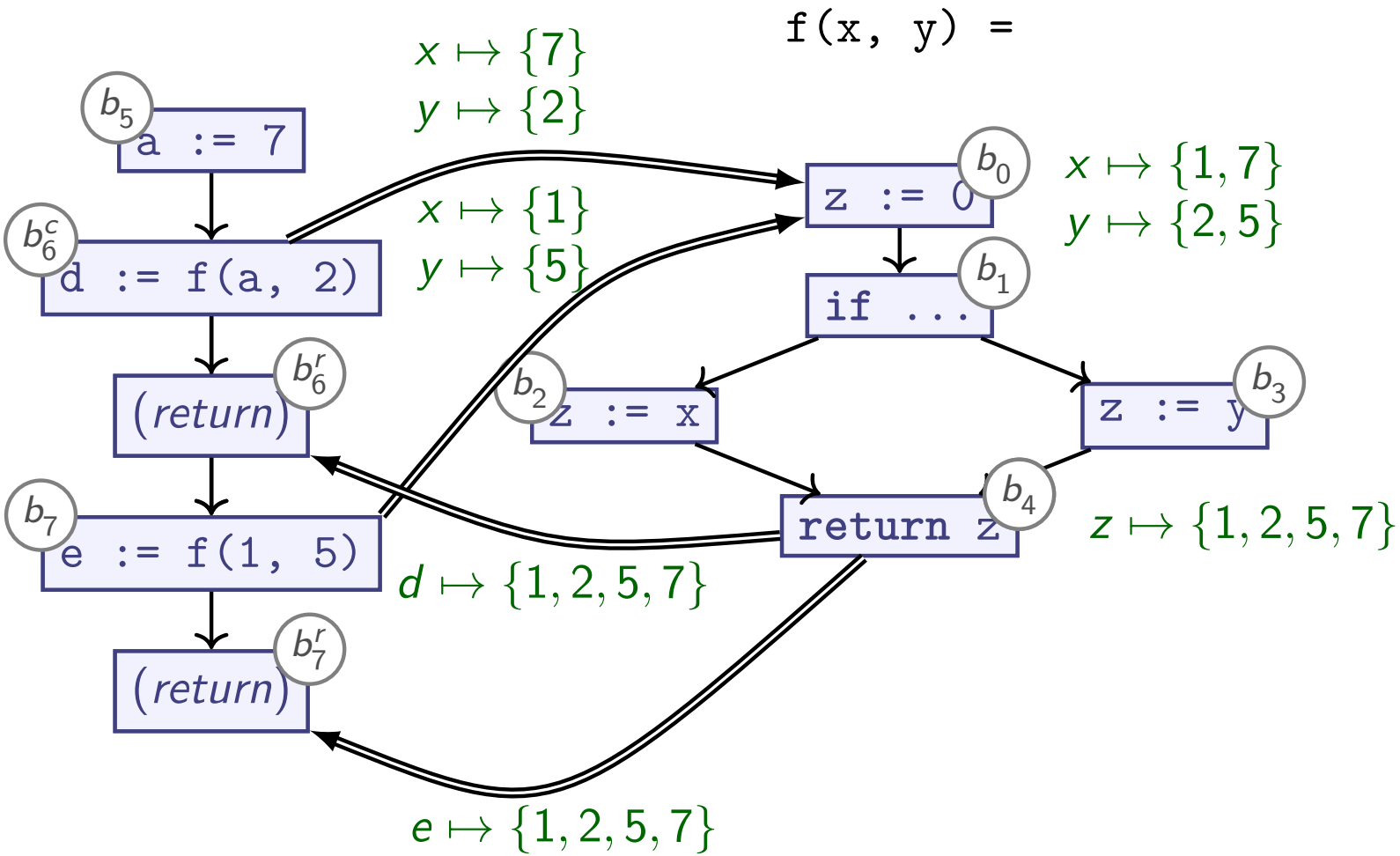
Works rather straightforwardly!

Inter-Procedural Control Flow Graph



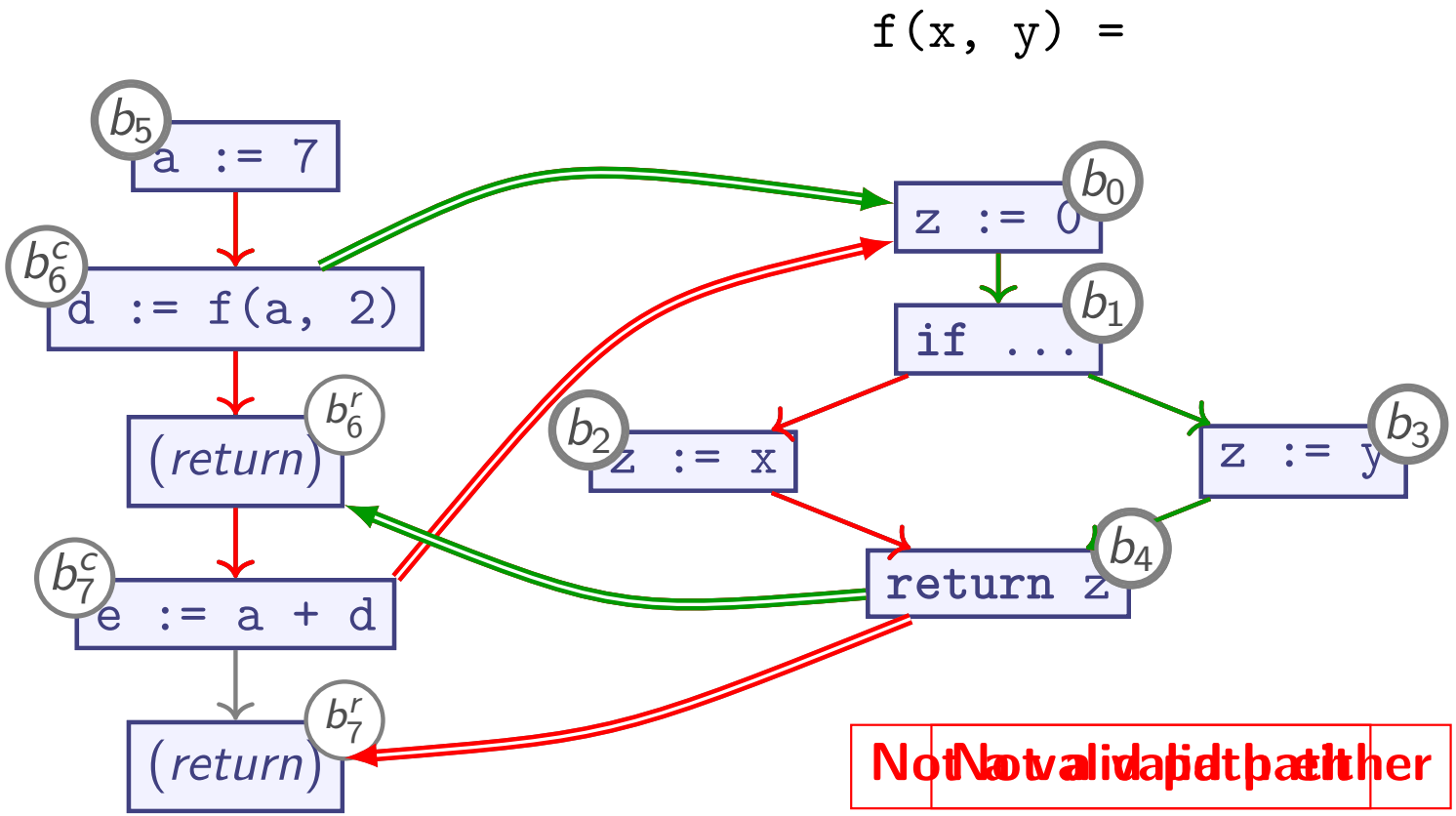
- Split call sites b_x into *call* (b_x^c) and *return* (b_x^r) nodes
- Intra-procedural edge $b_x^c \longrightarrow b_x^r$ carries environment/store
- Inter-procedural edge ($\Longrightarrow$):
  - Call site $\Longrightarrow$ callee: substitutes parameters
  - Call site $\Longleftarrow$ return: substitutes result
  - Otherwise like intra-procedural data flow edge

A Naïve Inter-Procedural Analysis



Imprecision!

Valid Paths



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

Context-site interprocedural analyses consider only valid paths

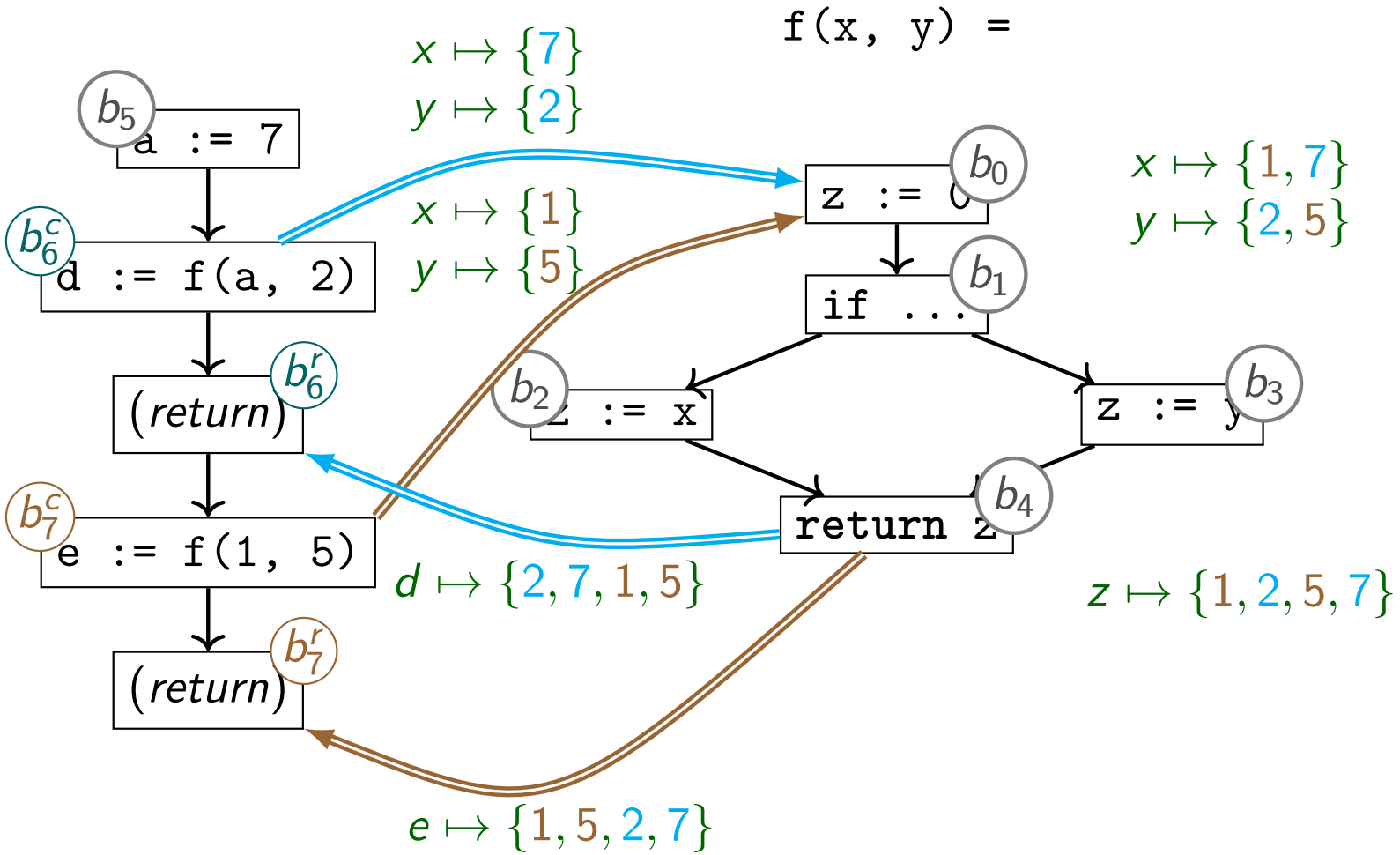
Summary

- **Intraprocedural** Analysis:
  - Considers one subroutine at a time
  - Calls to other subroutines treated as "worst-case" (e.g., $\top$ for dataflow analysis)
- **Interprocedural** Analysis:
  - Analyses calls to subroutines
  - For Dataflow analysis: uses **Interprocedural CFG** (ICFG)
    - ICFG represents subroutine calls as two nodes: **call** and **return**
    - Special Call/Return edges caller $\Leftrightarrow$ callee
    - Naïve interpretation of ICFG call/return edges "spills" analysis results across call sites

Interprocedural Data Flow Analysis

- **Call-site sensitive**
  - *Specialise* abstractions to different call sites
- **Call-site insensitive**
  - Use same abstraction for each call site
  - Examples for dataflow analysis:
    - Treat ICFG call/return edges like "regular" CFG edges
    - Use special transfer function everywhere
      E.g. for built-in operations (add, sub, print, . . .)

Call-Site Insensitive Analysis

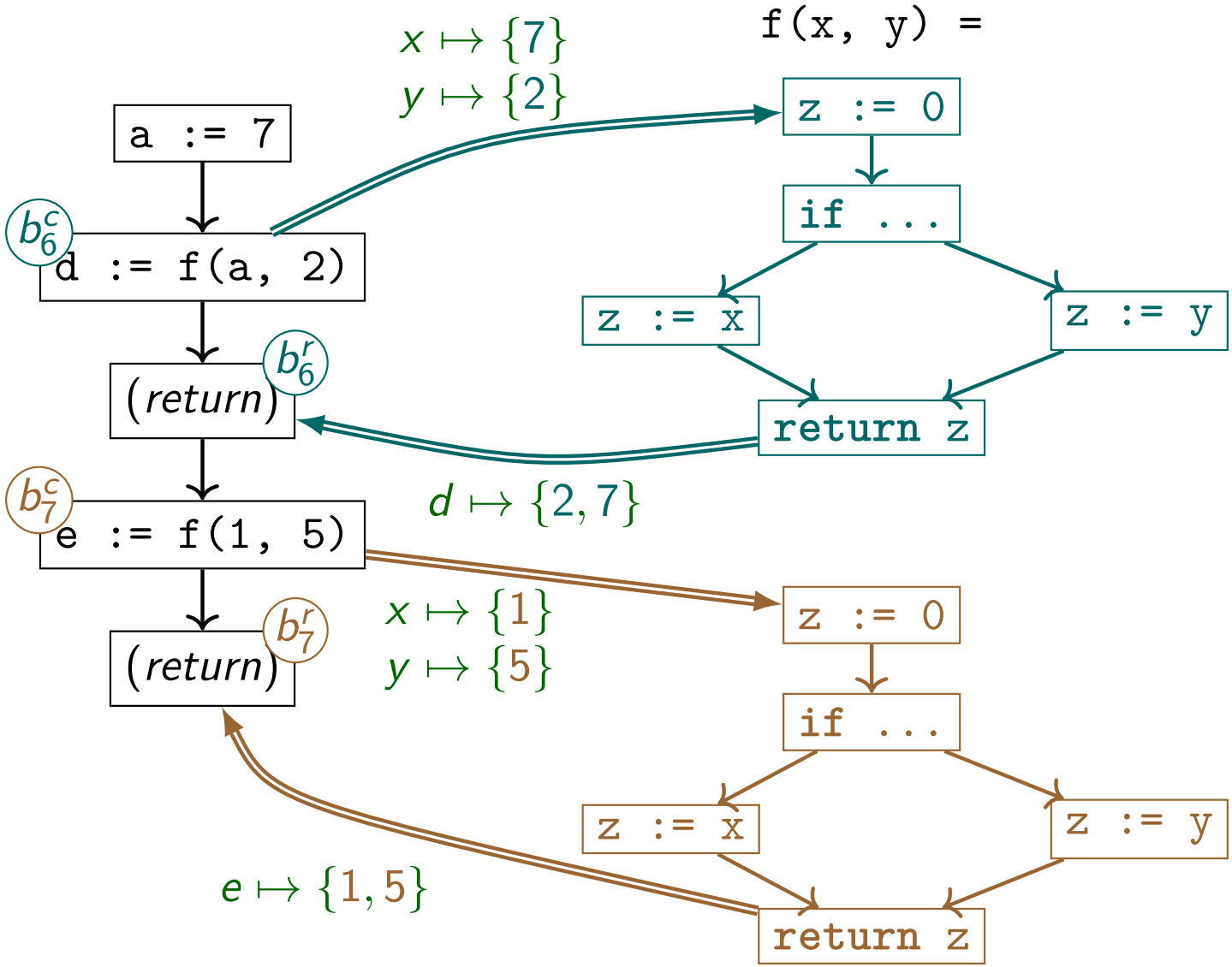


***Call-site insensitive*: analysis merges all callers to `f()`**

Precise Interprocedural Dataflow

- Precision via one of:
  1. **Inlining** or **AST cloning**
  2. Call Strings
  3. Procedure Summaries

Inlining

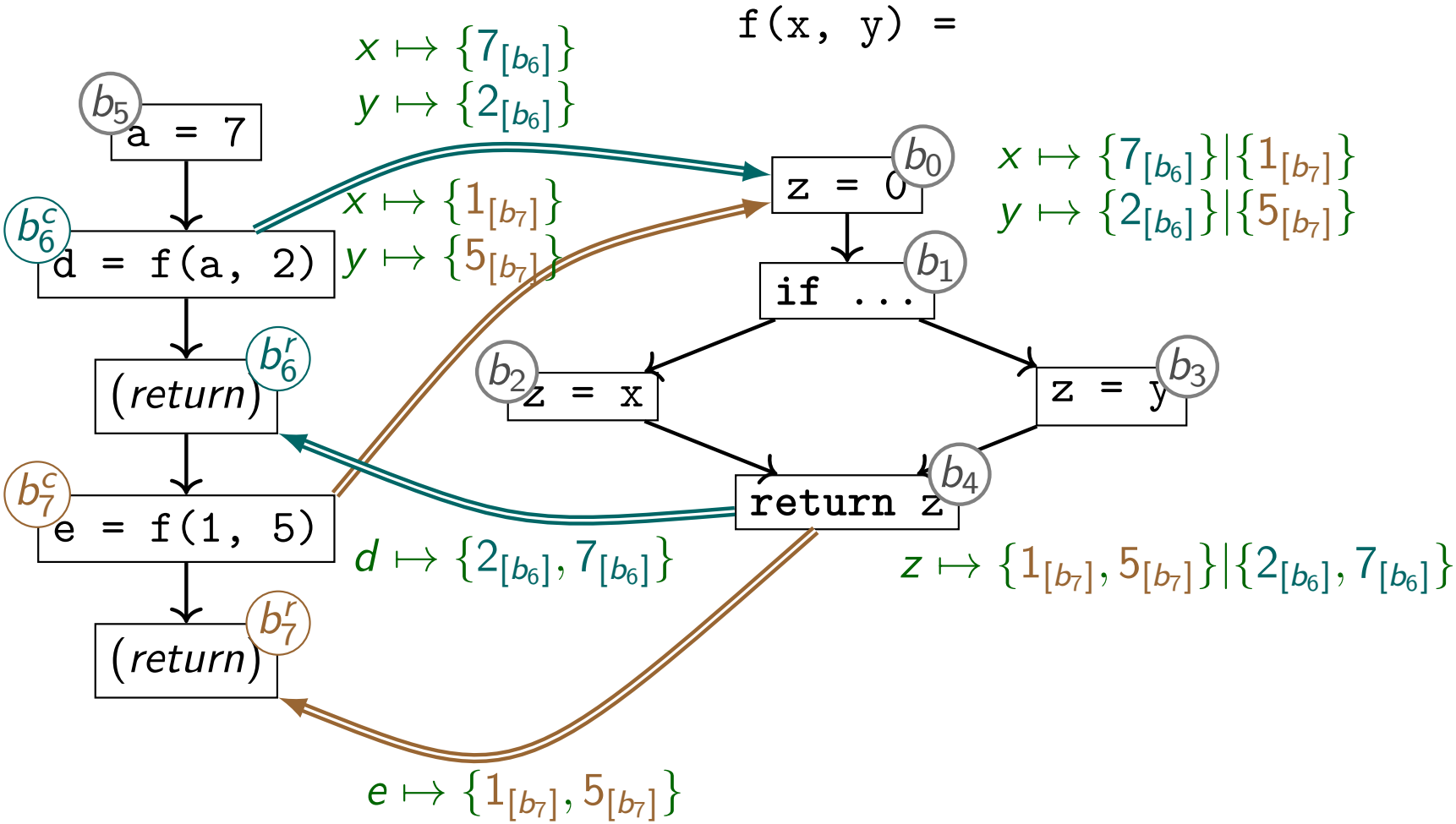


Clone subroutine IRs for each *calling context*

Precise Interprocedural Dataflow

- Precision via one of:
  1. Inlining or AST cloning
  2. **Call Strings**
  3. Procedure Summaries

Call Strings of Length 1



Degrees of Call-Site Sensitivity

- We used *call strings* to make call sites explicit:
  - $[b_6]$ in $2_{[b_6]}$
- "Strings" because this idea generalises:
  - Can keep track of *multiple* callers
  - Example: *2-call-site sensitivity*: $[b_0, b_6]$ vs $[b_1, b_6]$

Teal

```
fun g(y:  int):  int = { return y }
fun f(x:  int):  int = {
  return g(x) // b6
       + g(5); // b7
}
...
  f(1); // b0
  f(2); // b1
```

Must bound length of call strings to ensure termination

Summary

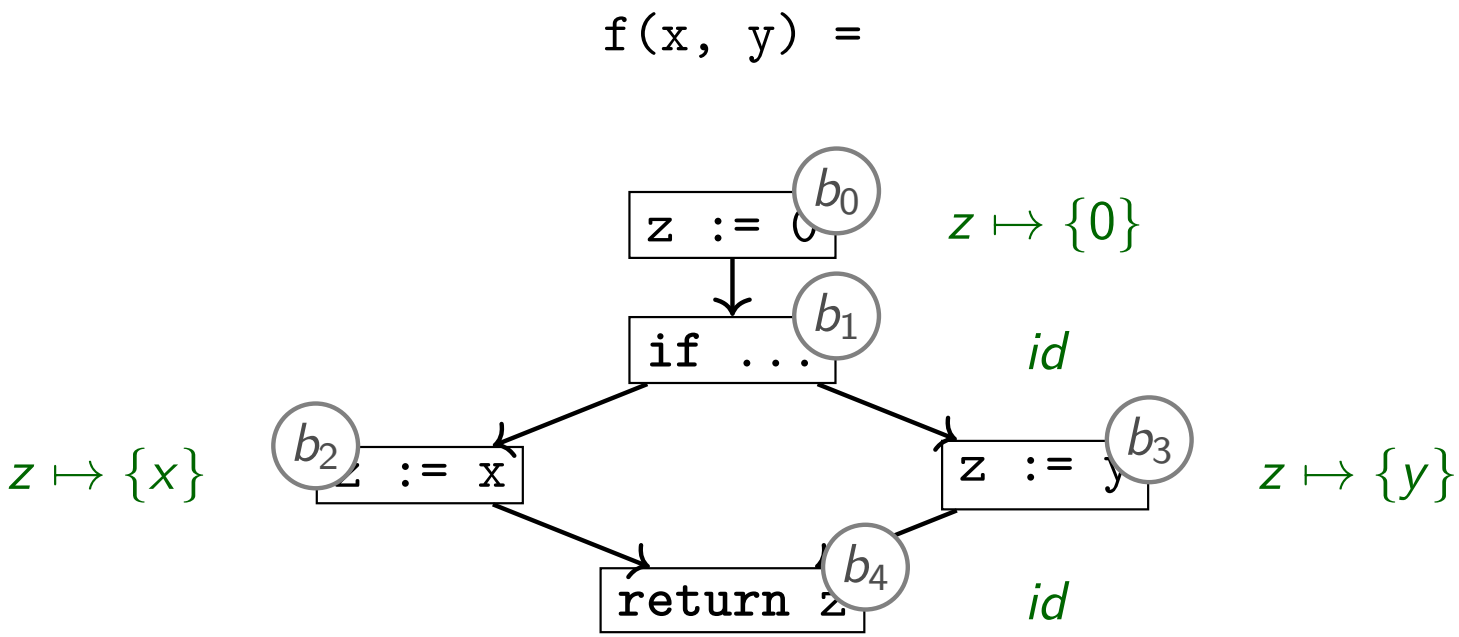
Strategies for call-site sensitive analysis:

- **Inlining**
  - Copy subroutine bodies for each caller
  - Performance cost
  - Recursion: fall back to $\top$
- **Call Strings**
  - Call string length:
    - Unbounded: Maximum precision, may not terminate with recursion
    - Bounded to length k: k degrees of call site sensitivity (speed/precision trade-off)

Precise Interprocedural Dataflow

- Precision via one of:
  1. Inlining or AST cloning
  2. Call Strings
  3. **Procedure Summaries**

Summarising Procedures



- *Compose* transfer functions:
  - $trans_{b_0} \circ trans_{b_1} = [z \mapsto 0]$
  - $trans_{b_0} \circ trans_{b_1} \circ trans_{b_2} = [z \mapsto \{x\}]$
  - $trans_{b_0} \circ trans_{b_1} \circ trans_{b_3} = [z \mapsto \{y\}]$
  - $trans_{b_0} \circ trans_{b_1} \circ (trans_{b_2} \sqcup trans_{b_3}) = [z \mapsto \{x, y\}]$
  - $trans_{b_0} \circ trans_{b_1} \circ (trans_{b_2} \sqcup trans_{b_3}) \circ trans_{b_4} = [z \mapsto \{x, y\}]$

Procedure Summaries vs Recursion

`f` calls `g` calls `h` calls `f`

- Reqiures additional analysis to identify who calls whom
- Compute summaries of mutually recursive functions together
- Recursive call edges analogous to loops
- Loops/recursion require fixpoint computation over function composition!

Procedure Summaries

- Composing transfer functions yields a combined transfer function for `f()`:

$$trans_f = [\textbf{return} \mapsto \{x, y\}]$$

- Use $trans_f$ as transfer function for `f()`, discard `f`'s body
- **Opportunities:**
  - Can yield compact subroutine descriptions
  - Can speed up call site analysis dramatically
- **Challenges:**
  - More complex to implement
  - Loops / recursion are challenging
- **Limitations:**
  - Requires suitable representation for summary
  - Requires mechanism for abstracting and applying summary
  - Worst cases:
    - $trans_f$ is symbolic expression more complex than `f` itself

Summary

Making interprocedural dataflow precise:

- **Call-site sensitive approaches:**
  - *Inlining*
  - *Call strings*
- **Call-site insensitive approaches:**
  - *Procedure Summaries*
    - Precise + compact summaries only possible for distributive frameworks

Outlook

- More Interprocedural Analysis on Tuesday

```
class A {
    public void f() { ...}
}
class B extends A {
    @Override
    public void f() { ...}
}
...
A x = ...;
x.f(); // A.f or B.f?
```

https://cs.lth.se/EDAP15