



**LUND**  
UNIVERSITY

# EDAP15: Program Analysis

---

## DATA FLOW ANALYSIS 1 INTRODUCTION

**Christoph Reichenbach**



# Towards Practical Program Analysis

Teal-0	Imperative and Procedural
Teal-1	Minor extensions on Teal-0
Teal-2	
Teal-3	

- ▶ Teal : Multi-layered language to exhibit program analysis challenges
- ▶ Small enough for homework exercises
- ▶ Big enough to exhibit real challenges
- ▶ Errors in Teal programs trigger *failures*:
  - ▶ Build analyses to detect failures before they happen

# Teal-0: A Procedural Language

<i>module</i>	::=	$\langle \text{import} \rangle^* \langle \text{decl} \rangle^*$	<i>expr</i>	::=	$\langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle$   <b>not</b> $\langle \text{expr} \rangle$   $( \langle \text{expr} \rangle \langle \text{opttype} \rangle )$   $\langle \text{expr} \rangle [ \langle \text{expr} \rangle ]$   <b>id</b> $( \langle \text{actuals} \rangle^? )$   $[ \langle \text{actuals} \rangle^? ]$   <b>new</b> $\langle \text{type} \rangle ( \langle \text{expr} \rangle )$   <b>int</b>   <b>string</b>   <b>null</b>   <b>id</b>
<i>import</i>	::=	<b>import</b> $\langle \text{qualified} \rangle ;$	<i>actuals</i>	::=	<i>expr</i>   <i>expr</i> , $\langle \text{actuals} \rangle$
<i>qualified</i>	::=	<i>id</i>   $\langle \text{qualified} \rangle :: \text{id}$	<i>binop</i>	::=	<b>+</b>   <b>-</b>   <b>*</b>   <b>/</b>   <b>%</b>   <b>==</b>   <b>!=</b>   <b>&lt;</b>   <b>&lt;=</b>   <b>&gt;=</b>   <b>&gt;</b>   <b>or</b>   <b>and</b>
<i>decl</i>	::=	$\langle \text{vardecl} \rangle ;$   <b>fun</b> <i>id</i> $( \langle \text{formals} \rangle^? ) \langle \text{opttype} \rangle = \langle \text{stmt} \rangle$	<i>stmt</i>	::=	$\langle \text{vardecl} \rangle$   $\langle \text{expr} \rangle ;$   $\langle \text{expr} \rangle := \langle \text{expr} \rangle ;$   $\langle \text{block} \rangle$   <b>return</b> $\langle \text{expr} \rangle ;$   <b>if</b> $\langle \text{expr} \rangle \langle \text{block} \rangle$ <b>else</b> $\langle \text{block} \rangle$   <b>if</b> $\langle \text{expr} \rangle \langle \text{block} \rangle$   <b>while</b> $\langle \text{expr} \rangle \langle \text{block} \rangle$
<i>vardecl</i>	::=	<b>var</b> <i>id</i> $\langle \text{opttype} \rangle$   <b>var</b> <i>id</i> $\langle \text{opttype} \rangle := \langle \text{expr} \rangle ;$			
<i>formals</i>	::=	<i>id</i> $\langle \text{opttype} \rangle$   <i>id</i> $\langle \text{opttype} \rangle , \langle \text{formal} \rangle$			
<i>opttype</i>	::=	<b>:</b> $\langle \text{type} \rangle$   $\epsilon$			
<i>type</i>	::=	<b>int</b>   <b>string</b>   <b>any</b>   <b>array</b> $[ \langle \text{type} \rangle ]$			
<i>block</i>	::=	$\{ \langle \text{stmt} \rangle^* \}$			

# Teal-0: Example

## Teal

```
var v := [0, 0];  
print(v);  
if z {  
    v[0] := 2;  
    v := null;  
}  
v[0] := 1;
```

# A New Analysis Challenge

## Teal

```
var x := [0, 0];  
print(x);    // A  
if z {  
    x[0] := 2; // B  
    x := null;  
}  
x[0] := 1;   // C
```

- ▶ Analyse: Can there be a *failure* at B or C?
- ▶ Must distinguish between x at A vs. x at B and C
- ▶ Need to model flow of information: **Flow-Sensitive Analysis**
- ▶ Type analysis is *not Flow-Sensitive* (normally)

Need analysis that can represent *data flow* through program

# Evaluation Order

## Teal-0

```
fun p(a) = { print(a); return 1; }  
p(p(0) + p(1));
```

## Teal-0 with explicit order

```
var tmp1 := p(0);  
var tmp2 := p(1);  
var tmp3 := tmp1 + tmp2;  
var tmp4 := p(tmp3);
```

## Java or C or C++

```
// Many challenging constructions:  
a[i++] = b[i > 10 ? i-- : i++] + c[f(i++, --i)];
```

**Every analysis must remember the evaluation order rules!**

# Eliminating Nested Expressions

- ▶ No nested expressions
  - ⇒ Evaluation order is explicit
  - ⇒ Fewer patterns to analyse
- ▶ All intermediate results have a name
  - ⇒ Easier to 'blame' subexpressions for errors
    - ▶ Names might be represented pointers in the implementation
- ▶ We still have nested statements

# Multiple Paths

## Teal

```
v := new array[int](1);  
if condition {  
    v := null;  
} else {  
    print(v);  
}  
v[0] := 1;
```

## Teal

```
v := new array[int](1);  
while condition {  
    v := null;  
}  
v[0] := 1;
```

Need to reason about the order of execution of *statements*, too



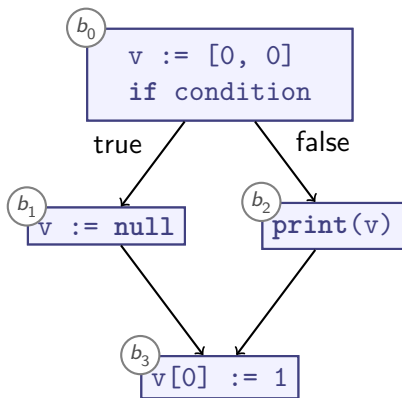
# Summary

- ▶ Understanding variable updates requires **Flow-Sensitive Analysis**
- ▶ Type analysis is *not* flow sensitive
- ▶ “Flow” is complicated, influenced by:
  - ▶ Expression evaluation order
  - ▶ Short-circuit evaluation
  - ▶ Statement execution order
- ▶ Best analysed with special intermediate representation:
  - ▶ Flatten nested expressions
  - ▶ Introduce temporary variables as needed
  - ▶ ... do something about statement execution? (up next!)

# Control-Flow Graphs (CFGs)


## Teal

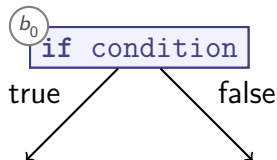
```
var v := [0, 0];  
if condition {  
    v := null;  
} else {  
    print(v);  
}  
v[0] := 1;
```



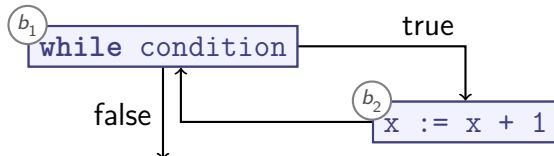
**Control Flow Graphs encode statement execution order**

# Control-Flow-Graphs

- ▶ Encode statement order by *nodes*  and edges  $\rightarrow$
- ▶ *Multiple* outgoing edges (branches): Add label:

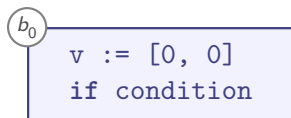


- ▶ Uniform representation for control statements:

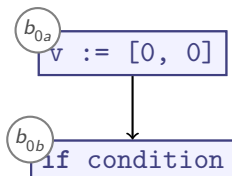


# Basic Blocks

Can group statements into **Basic Blocks** or keep them separate:



Basic Block



- ▶ A **Basic Block** is a sequence of statements
- ▶ Last statement is *always* return, branch, or jump
- ▶ Other statements are *never always* return, branch, or jump
- ▶ Usually faster to process

# Summary

- ▶ Different **Intermediate Representations** (IRs) to pick
- ▶ Usually eliminate nested expressions
  - ▶ Make evaluation order explicit
- ▶ **Control-Flow Graph** (CFG):
  - ▶ Represent control flow as **Blocks** and **Control-Flow Edges**
  - ▶ Edges represent control flow, **labelled** to identify conditionals
  - ▶ Blocks can be single statements or **Basic Blocks**
    - ▶ Basic blocks are sequences of statements without branches

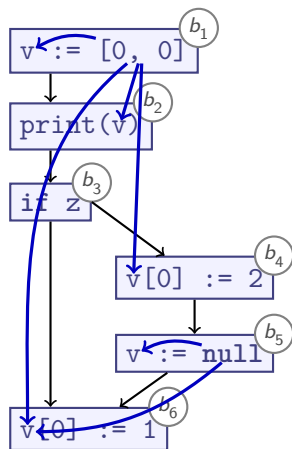
# Control Flow

Understanding **data flow** requires understanding control flow:

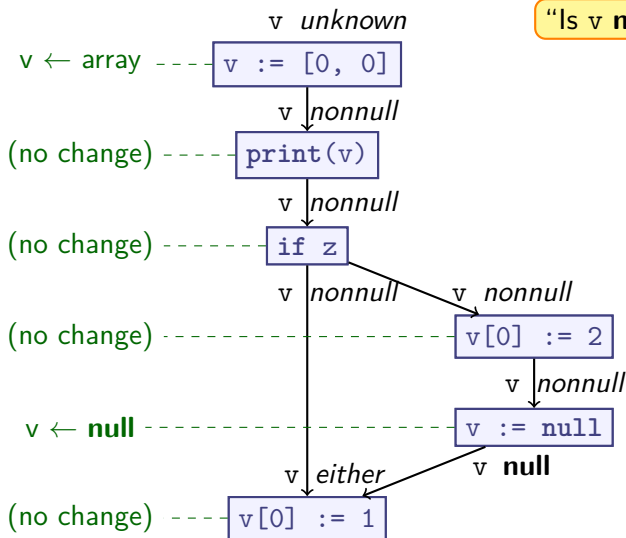
## Teal

```
var v := [0, 0];  
print(v);  
if z {  
    v[0] := 2;  
    v := null;  
}  
v[0] := 1;
```

→ Control flow  
→ Data flow



# Basic Ideas of Data Flow Analysis



# Another Analysis

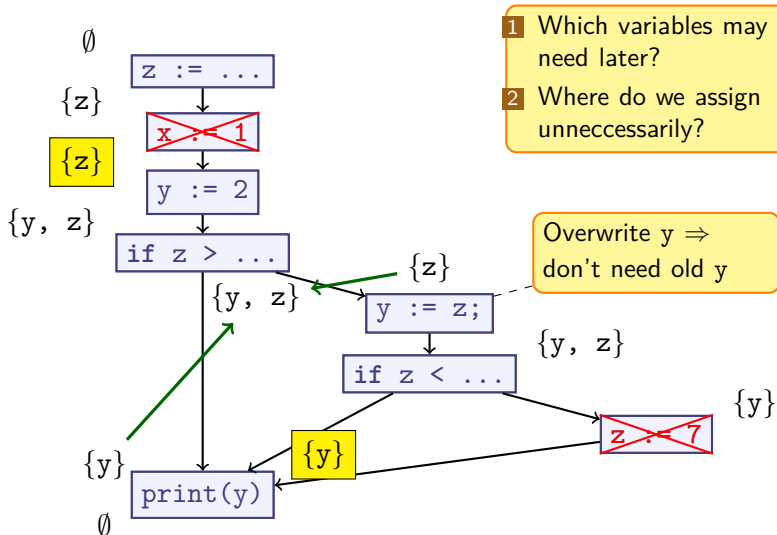
## Teal

```
z := ...  
x := 1;  
y := 2;  
if z > ... {  
    y := z  
    if z < ... {  
        z := 7  
    }  
}  
print(y);
```

- Which assignments are unnecessary?
- ⇒ Possible oversights / bugs  
(*Live Variables Analysis*)



# Unnecessary Assignments: Intuition

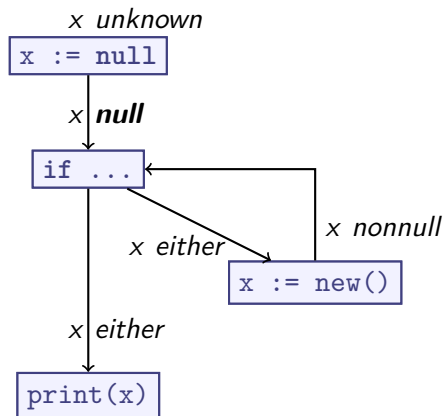


Analysis effective: found useless assignments to  $z$  and  $x$

# Observations

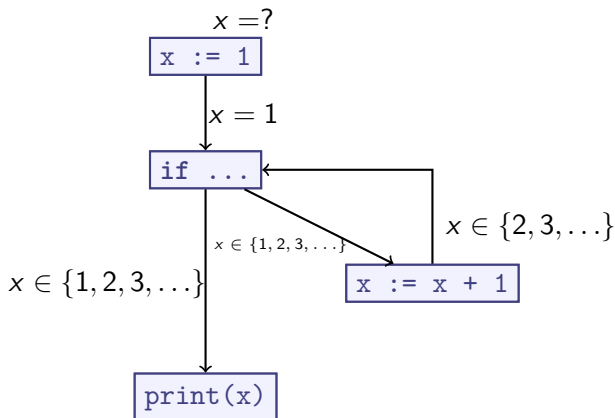
- 1 Data Flow analysis can be run *forward* or *backward*
- 2 May have to *join* results from multiple sources
- 3 Some analyses may need multiple “passes” (steps)

# What about Loops? (1/2)



- Analysis: *Null Pointer Dereference*
- Stop when we're not learning anything new any more
- Works fine

# What about Loops? (2/2)



- Analysis: *Reaching Definitions*

**We need to bound repetitions!**

# Summary: Data-Flow Analysis (Introduction)

- ▶ Data flow depends on *control flow*
- ▶ Data flow analysis examines how variables or other program state change across control-flow edges
- ▶ May have to join multiple results
- ▶ Can run *forward* or *backward* relative to control flow edges
- ▶ Handling loops is nontrivial

# Engineering Data Flow Algorithms

## 1 General Algorithm

- ▶ Keep updating until nothing changes

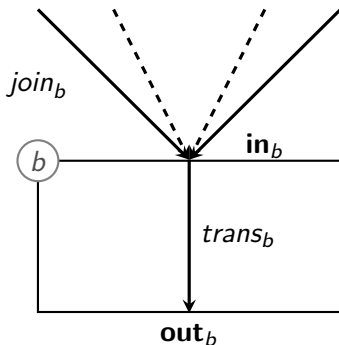
## 2 Termination

- ▶ Assumption: Operate on Control Flow Graph
- ▶ Theory: Ensure termination

## 3 (Correctness)

# Data Flow Analysis on CFGs

- ▶  $\mathbf{in}_b$ : knowledge at entrance of basic block  $b$
- ▶  $\mathbf{out}_b$ : knowledge at exit of basic block  $b$
- ▶  $\mathbf{join}_b$ : combines all  $\mathbf{out}_{b_i}$  for all basic blocks  $b_i$  that flow into  $b$   
“Join Function”
- ▶  $\mathbf{trans}_b$ : updates  $\mathbf{out}_b$  from  $\mathbf{in}_b$   
“Transfer Function”



# Characterising Data Flow Analyses

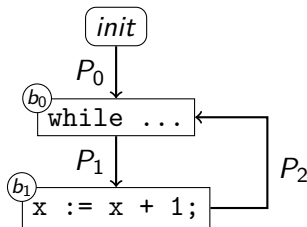
## Characteristics:

- ▶ *Forward* or *backward* analysis
- ▶  $L$ : Abstract Domain (the ‘analysis domain’)
- ▶  $trans_b : L \rightarrow L$
- ▶  $join_b : L \times L \rightarrow L$

**Require properties of  $L$ ,  $trans_b$ ,  $join_b$  to ensure termination**



# Limiting Iteration



- ▶ Does the following ever stop changing:

$$\mathbf{in}_{b_0} = \mathit{join}_{b_0}(P_0, P_2)$$

- ▶ Intuition: we keep generalising information
  - ▶ *Growth limit*: bound amount of generalisation
  - ▶ Make sure  $\mathit{join}_b$ ,  $\mathit{trans}_b$  never throw information away

**Eventually, either nothing changes or we hit growth limit**

# Ordering Knowledge

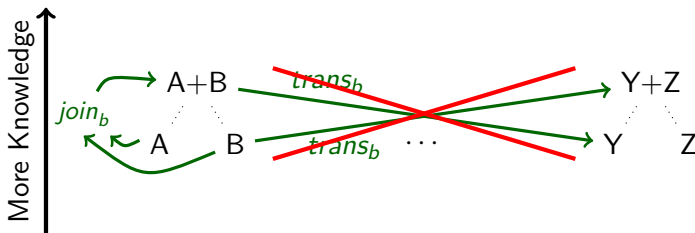
$$B \sqsupseteq A$$



- ▶  $B$  describes at least as much knowledge as  $A$
- ▶ Either:
  - ▶  $A = B$  (i.e.,  $A \sqsupseteq B \sqsupseteq A$ ), or
  - ▶  $B$  has strictly more knowledge than  $A$

# Intuition: Knowing Less, Knowing More

## Structure of $L$ :

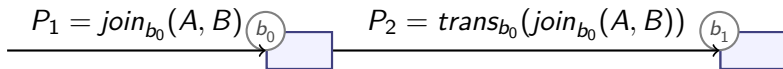


- ▶  $join_b$  must not lose knowledge
  - ▶  $join_b(A, B) \sqsupseteq A$
  - ▶  $join_b(A, B) \sqsupseteq B$
- ▶  $trans_b$  must be *monotonic* over amount of knowledge:

$$x \sqsupseteq y \implies trans_b(x) \sqsupseteq trans_b(y)$$

- ▶ Introduce bound:  $\top$  means 'too much information'

# Aggregating Knowledge



- ▶ Interplay between  $\text{trans}_b$  and  $\text{join}_b$  helps preserve knowledge
- ▶  $\text{join}_b(A, B) \sqsupseteq A$ :  
As we add knowledge,  $P_1$  either
  - ▶ Stays the same
  - ▶ Increases knowledge
- ▶ Monotonicity of  $\text{trans}_b$ : If  $P_1$  goes up, then  $P_2$  either
  - ▶ Stays the same
  - ▶ Increases knowledge

⇒ At each node, we either stay equal or grow

**Now we must only set a growth limit. . .**

# Ascending Chains

- ▶ A (possibly infinite) sequence  $a_0, a_1, a_2, \dots$  is an *ascending chain* iff:

$$a_k = a_{k+1} = \dots$$

$$a_i \sqsubseteq a_{i+1} \text{ (for all } i \geq 0)$$

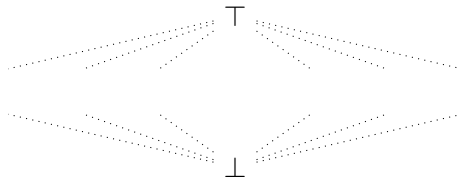
- ▶ *Ascending Chain Condition*:
  - ▶ For every ascending chain  $a_0, a_1, a_2, \dots$  in abstract domain  $L$ :
  - ▶ there exists  $k \geq 0$  such that:

$$a_k = a_{k+n} \text{ for any } n \geq 0$$

$a_3$   
|  
 $a_2$   
|  
 $a_1$   
|  
 $a_0$

**ACC is formalisation of growth limit**

# Top and Bottom



- ▶ *Convention:* We introduce two distinguished elements:
  - ▶ **Top:**  $\top: A \sqsubseteq \top$  for all  $A$
  - ▶ **Bottom:**  $\perp: \perp \sqsubseteq A$  for all  $A$
- ▶ Since  $join_b(A, B) \sqsupseteq A$  and  $join_b(A, B) \sqsupseteq B$ :
  - ▶  $join_b(\top, A) = \top = join_b(A, \top)$
  - ▶  $join_b(\perp, A) \sqsupseteq A \sqsupseteq \perp$ 
    - ▶ In practice, it's safe and simple to set:  
 $join_b(\perp, A) = A = join_b(A, \perp)$
- ▶ *Intuition:*
  - ▶  $\top$ : means 'contradictory / too much information'
  - ▶  $\perp$ : means 'no information known yet'

# Summary

- ▶ Designing a *Forward* or *backward* analysis:

- ▶ Pick **Abstract Domain**  $L$

- ▶ Must be **partially ordered** with  $(\sqsubseteq) \subseteq L \times L$ :

$A \sqsubseteq B$  iff  $A$  'knows' at least as much as  $B$

- ▶ Unique top element  $\top$
  - ▶ Unique bottom element  $\perp$

- ▶  $trans_b : L \rightarrow L$

- ▶ Must be *monotonic*:

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

- ▶  $join_b : L \times L \rightarrow L$  must produce an *upper bound* for its parameters:

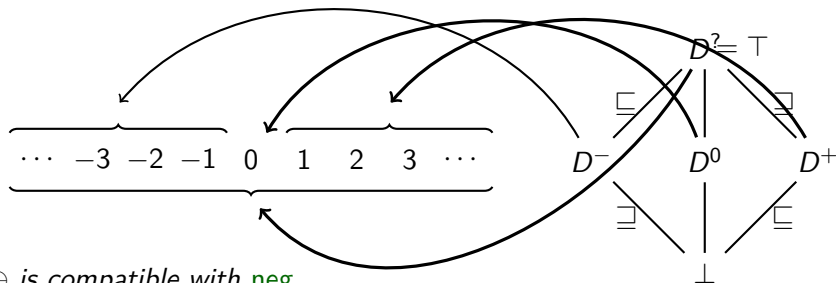
- ▶  $join_b(A, B) \sqsubseteq A$

- ▶  $join_b(A, B) \sqsubseteq B$

- ▶ Satisfy **Ascending Chain Condition** to ensure termination

- ▶ Easiest solution: make  $L$  finite

# Abstract Domains Revisited



$\ominus$  is compatible with **neg**

$$\begin{aligned}
 \ominus \perp &= \perp \\
 \ominus D^0 &= D^0 \\
 \ominus D^+ &= D^- \\
 \ominus D^- &= D^+ \\
 \ominus D^? &= D^?
 \end{aligned}$$

$\ominus$  is monotonic (and  $\oplus$  extended with  $\perp$  is, too)



# Summary

- ▶ We could extend  $\{D^+, D^-, D^0, D^?\}$  to an Abstract Domain by adding  $\perp$

$$L_D = \{D^+, D^-, D^0, D^?, \perp\}$$

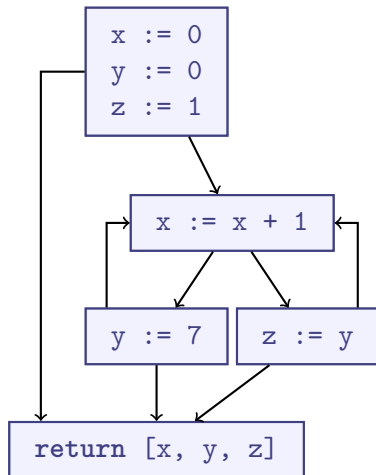
- ▶  $L_D$  is finite, so the DCC holds trivially
- ▶ Our *Transfer Functions*  $\ominus, \oplus$  are monotonic

# Example: Reaching Definitions

```
var x := 0;  
var y := 0;  
var z := 1;
```

```
while x < 5 {  
  x := x + 1;  
  if x >= 2 {  
    y := 7;  
  } else {  
    z := y;  
  }  
}
```

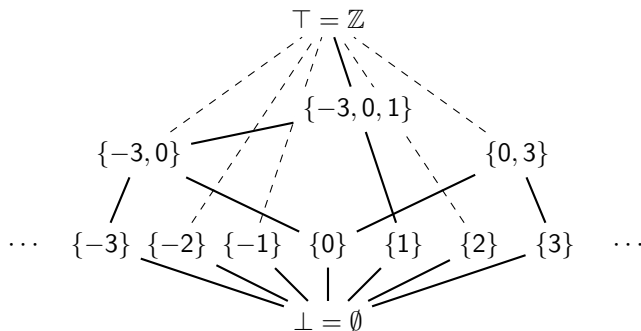
```
return [x, y, z];
```



**Reaching Definitions: What values are possible?**

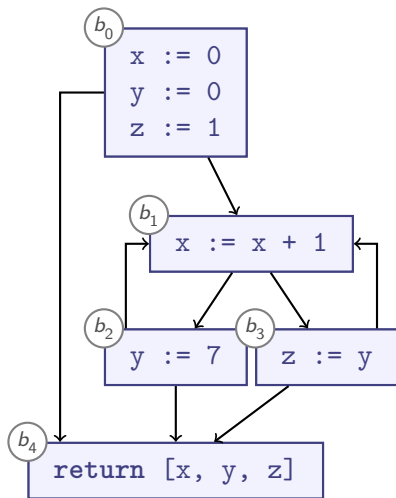
# Example: Reaching Definitions

Designing our abstract domain:



- ▶ Capture sets of up to 3 possible numbers
- ▶  $\top$ : More than 3 possible numbers
- ▶  $\perp$ :  $\emptyset$  (no possible numbers seen yet)
- ▶ Infinitely many elements, but finite height!

# Example: Control-Flow Graph

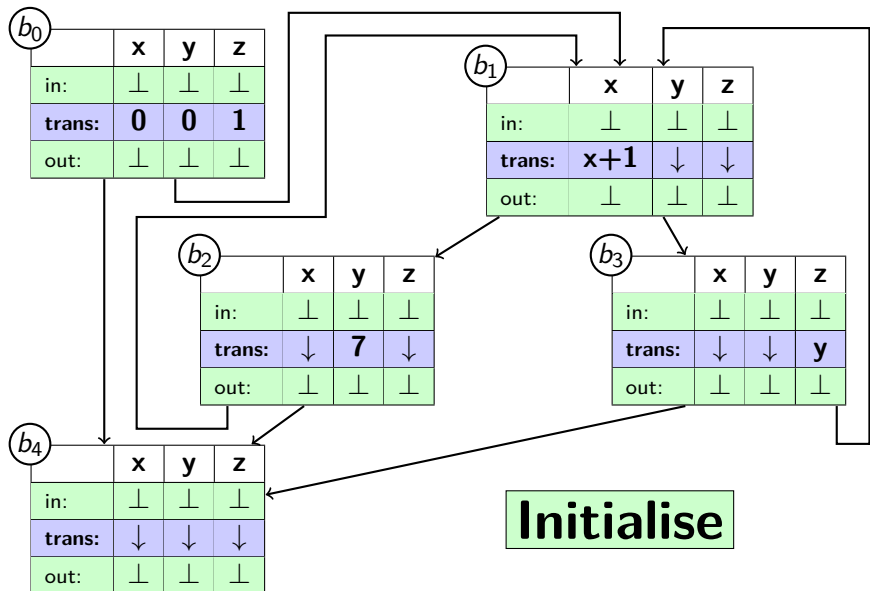


$b$	inputs	$trans_b$		
		$x$	$y$	$z$
$b_0$	$\emptyset$	0	0	1
$b_1$	$\{b_0, b_2, b_3\}$	$x + 1$	$y$	$z$
$b_2$	$\{b_1\}$	$x$	7	$z$
$b_3$	$\{b_1\}$	$x$	$y$	$y$
$b_4$	$\{b_0, b_2, b_3\}$	$x$	$y$	$z$

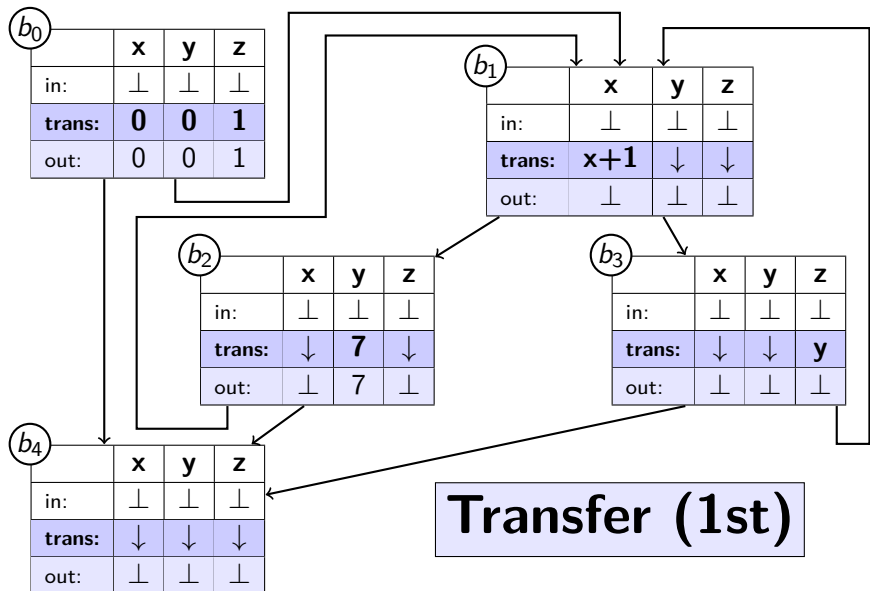
$$\begin{aligned} join_b = \text{let } j = \bigcup_{s \in \text{inputs}_b} s \\ \text{in } \begin{cases} j & \iff \#j \leq 3 \\ \top & \iff \#j > 3 \end{cases} \end{aligned}$$

This  $join_b$  builds union of input sets, or  $\top$  if that union has more than 3 elements

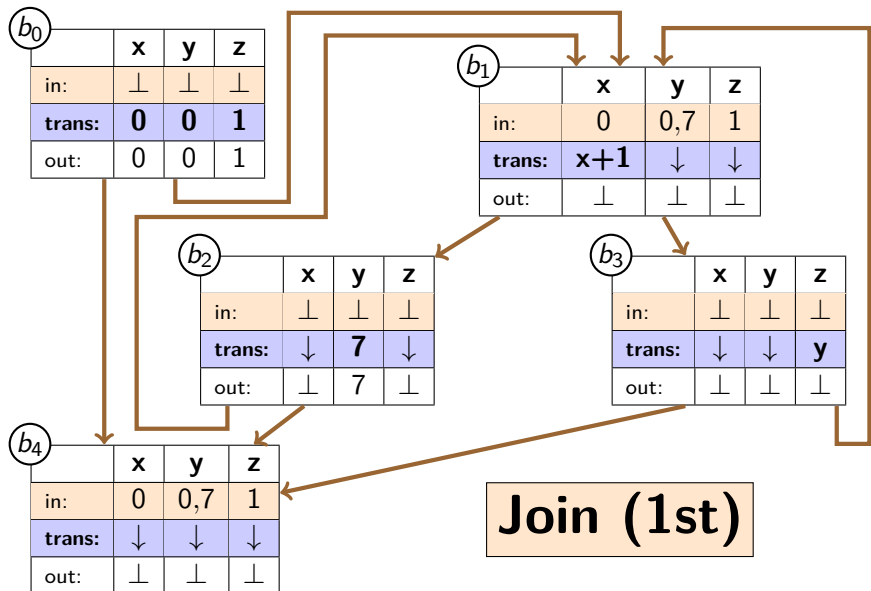
# Example: Computing the Fixpoint



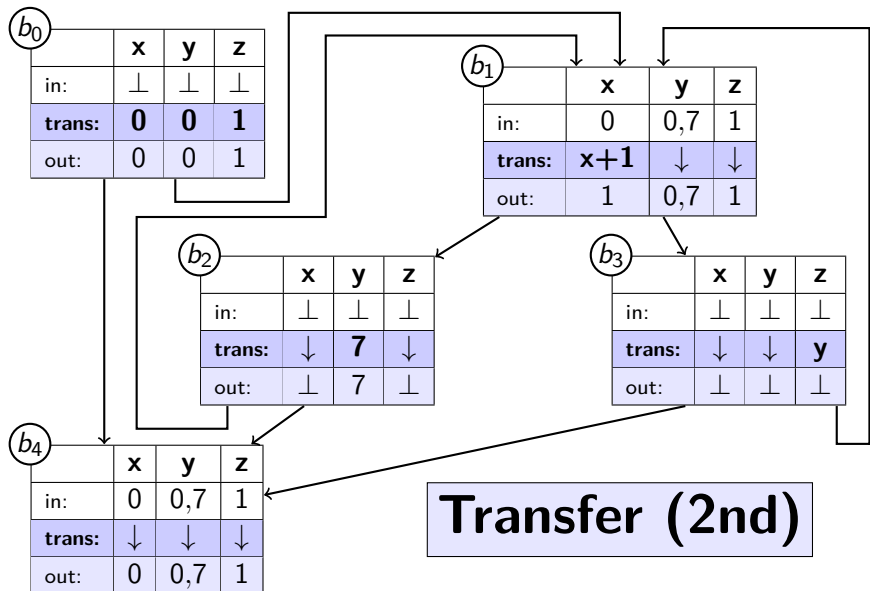
# Example: Computing the Fixpoint



# Example: Computing the Fixpoint

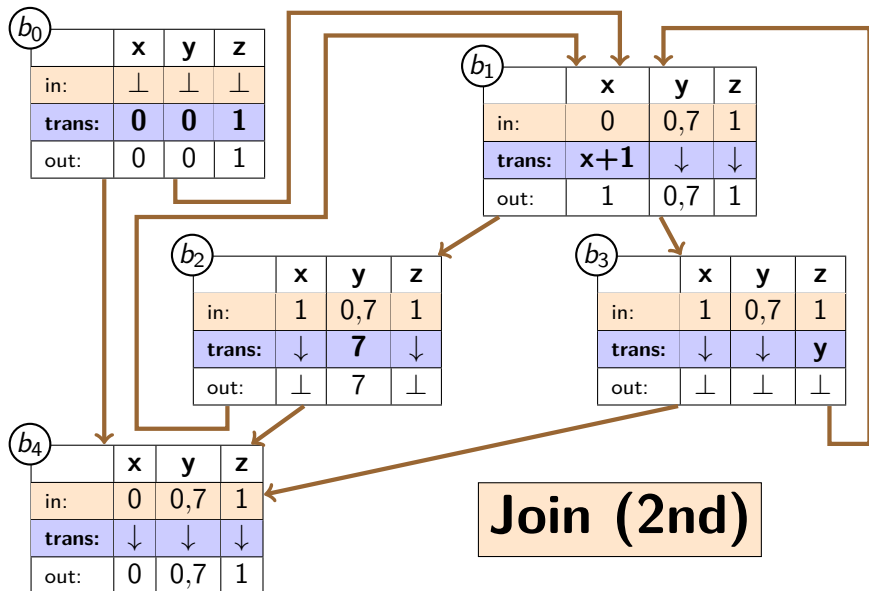


# Example: Computing the Fixpoint

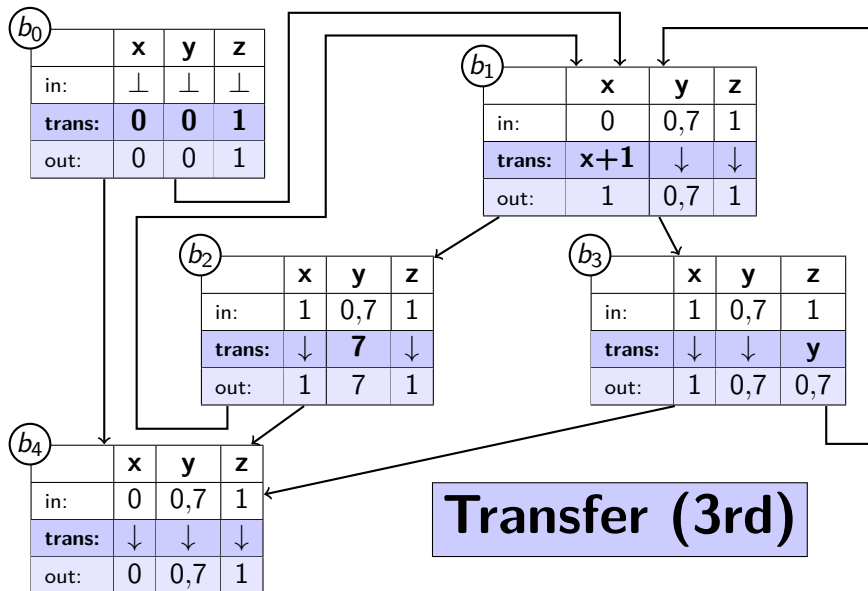




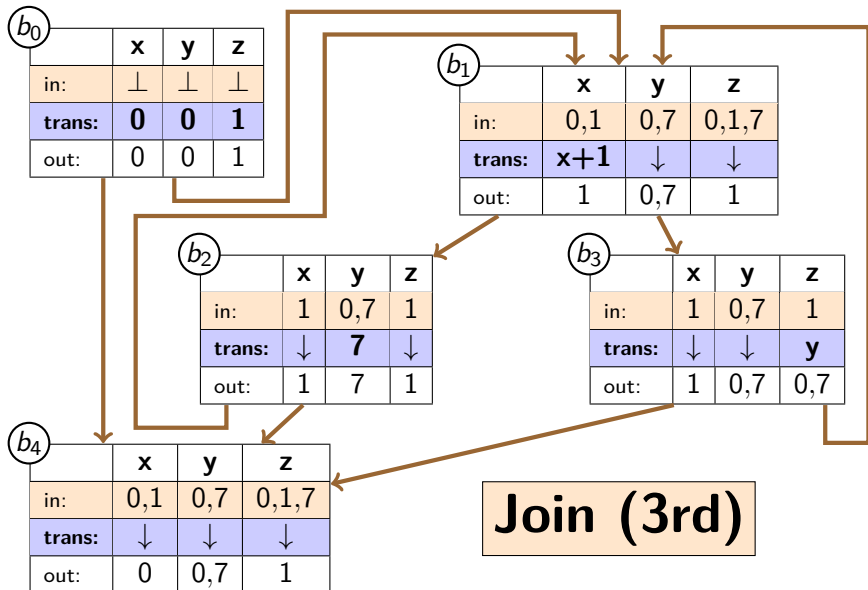
# Example: Computing the Fixpoint



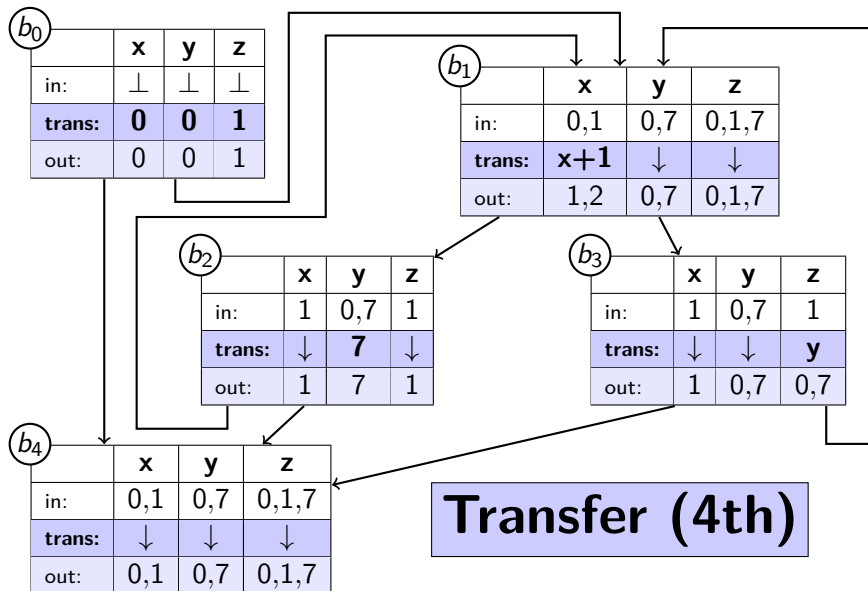
# Example: Computing the Fixpoint



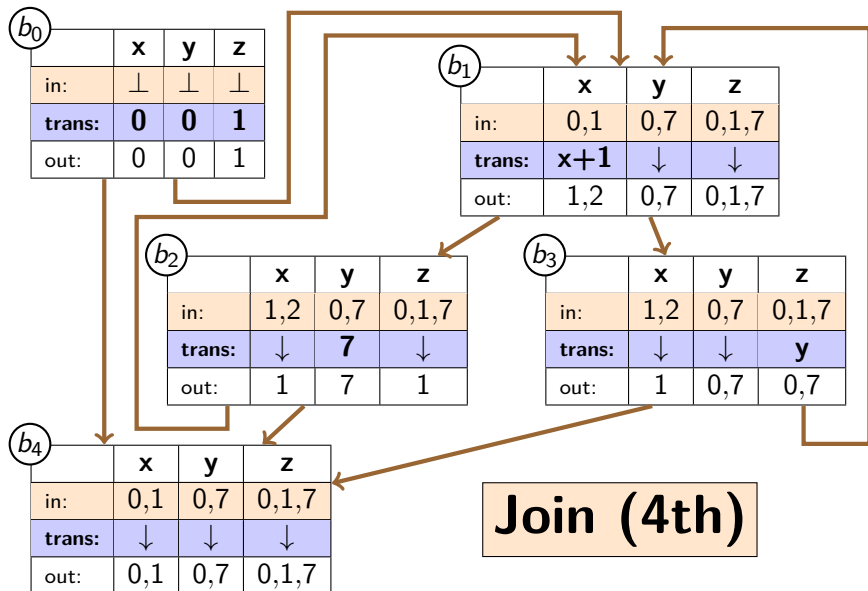
# Example: Computing the Fixpoint



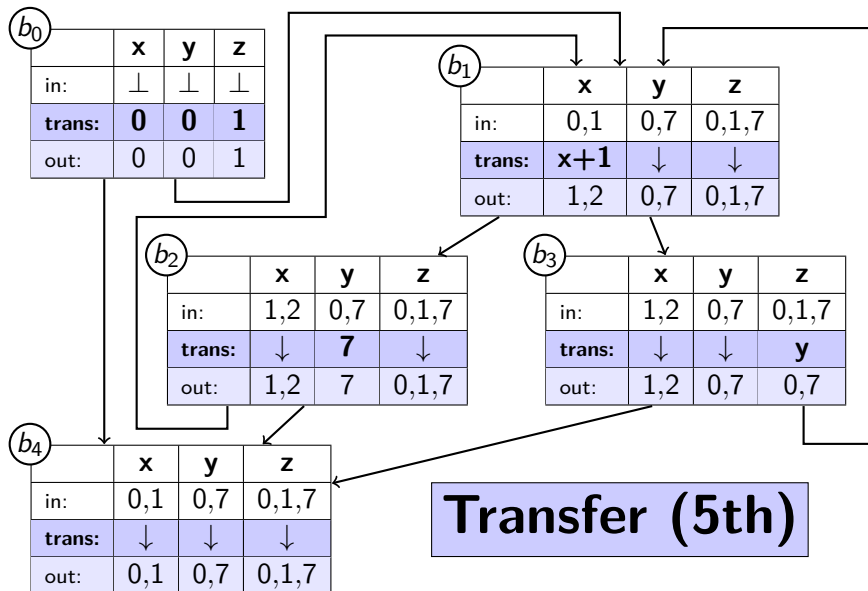
# Example: Computing the Fixpoint



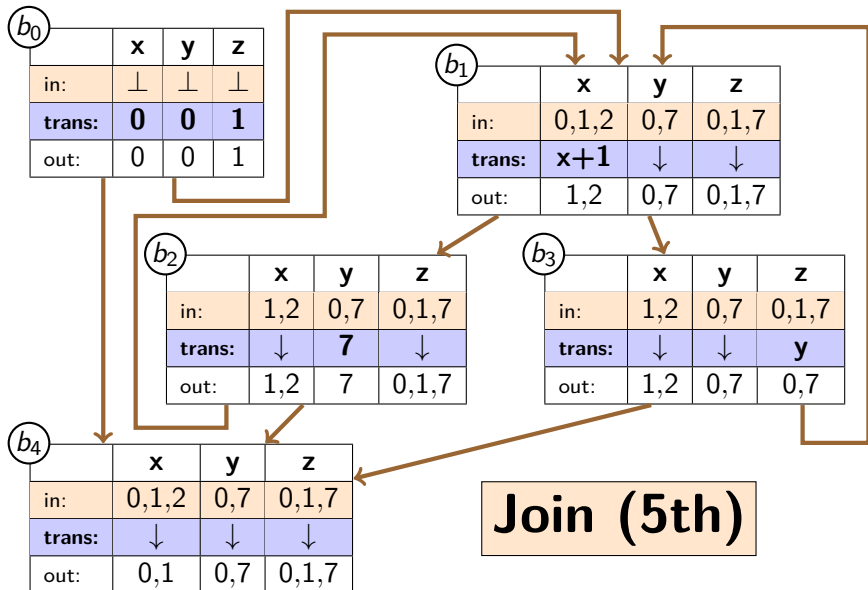
# Example: Computing the Fixpoint



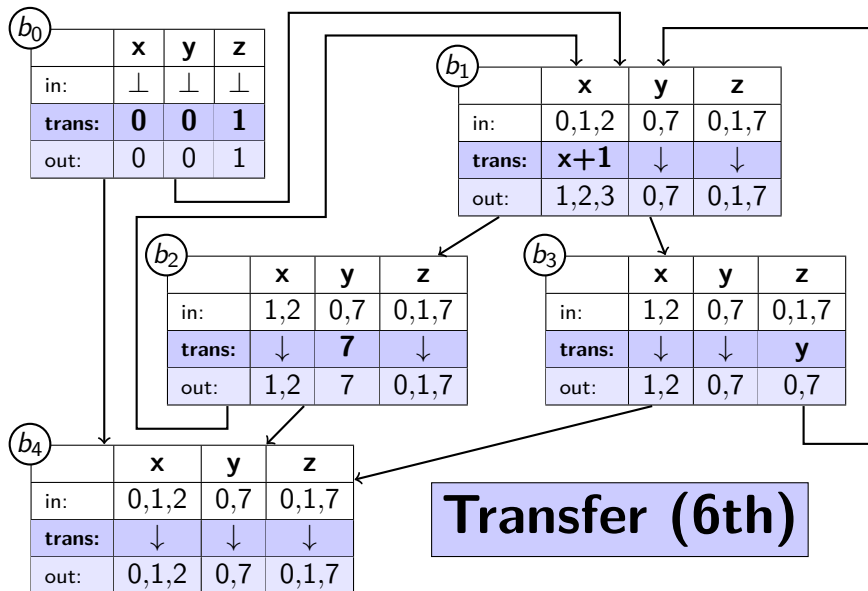
# Example: Computing the Fixpoint



# Example: Computing the Fixpoint

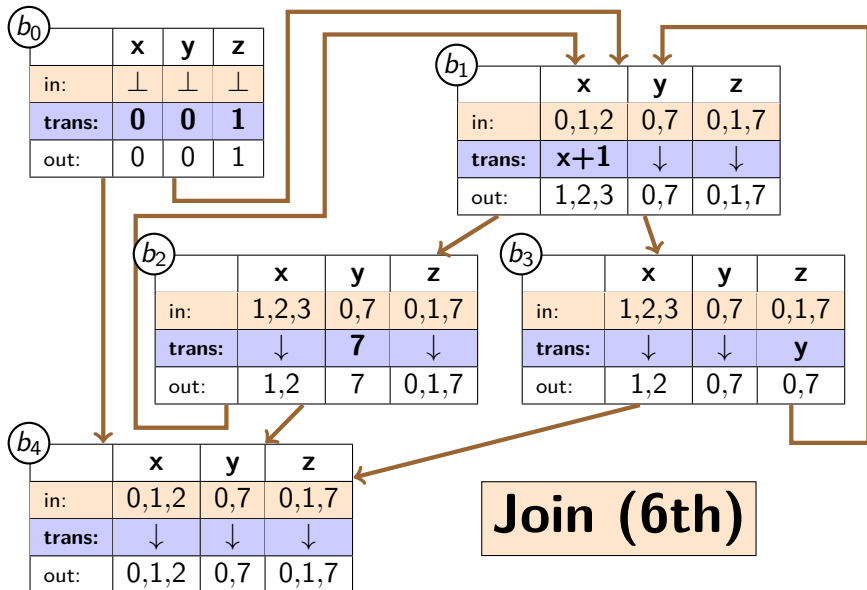


# Example: Computing the Fixpoint

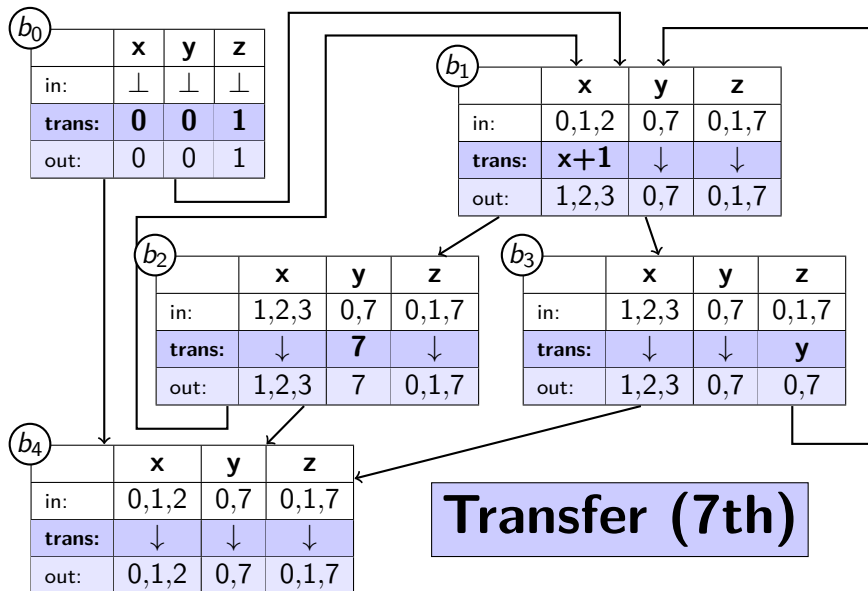




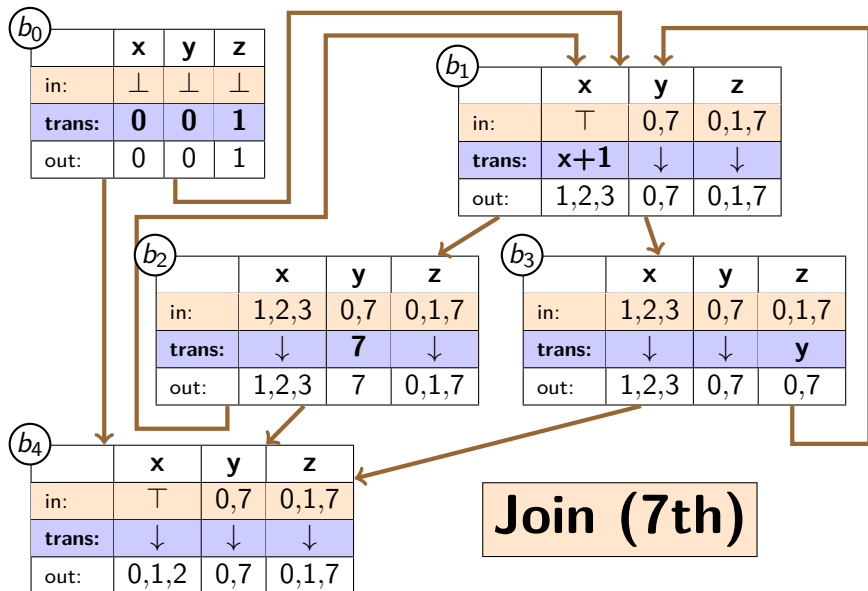
# Example: Computing the Fixpoint



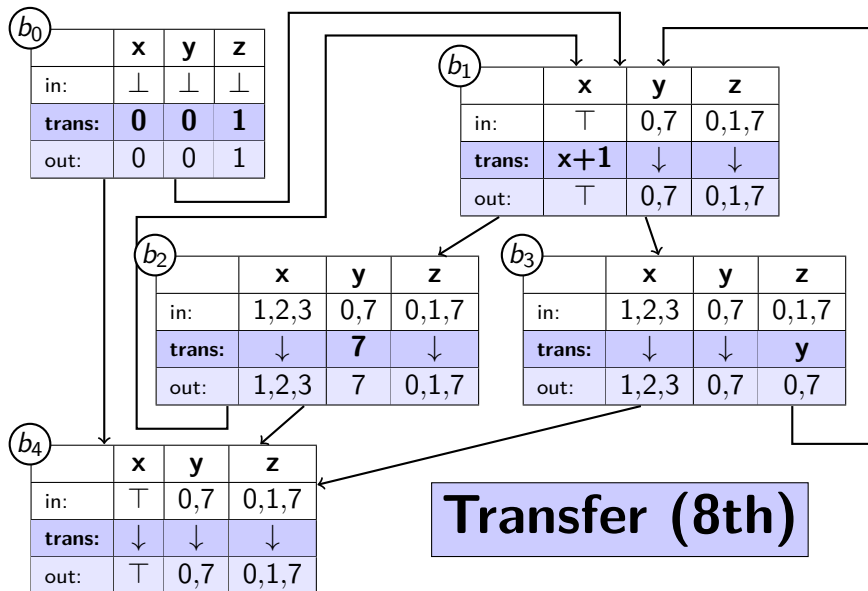
# Example: Computing the Fixpoint



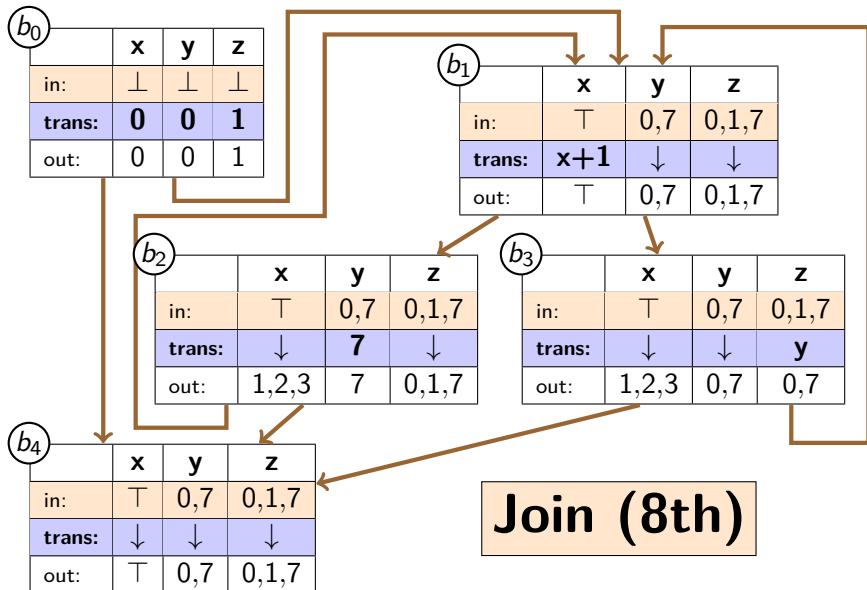
# Example: Computing the Fixpoint



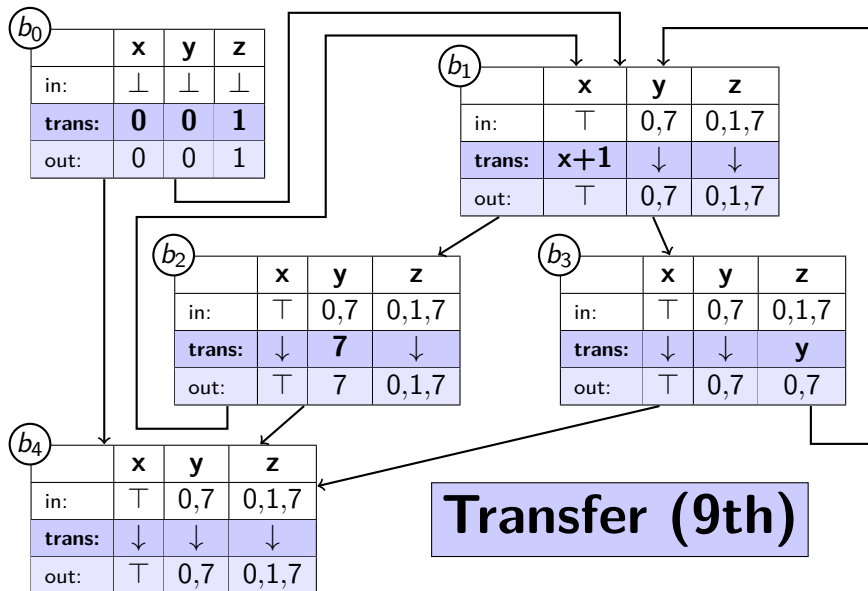
# Example: Computing the Fixpoint



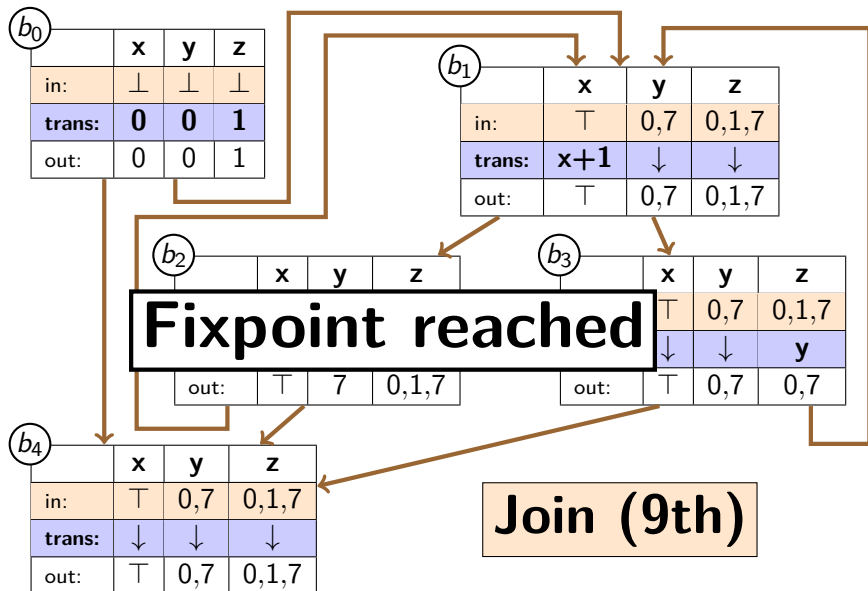
# Example: Computing the Fixpoint



# Example: Computing the Fixpoint



# Example: Computing the Fixpoint



# Example: Conclusion

```
var x := 0;
var y := 0;
var z := 1;

while x < 5 {
  x := x + 1;
  if x >= 2 {
    y := 7;
  } else {
    z := y;
  }
}

return [x, y, z];
```

- ▶ Applied abstract domain to three variables
- ▶ Reached fixpoint after 9 iterations
- ▶ Return values:
  - x :  $\top$  (unknown/any)
  - y : 0 or 7
  - z : 0 or 1 or 7
- ▶ Conservative approximation of reality
- ▶ Once x reached more than 3 values, algorithm gave up and went to  $\top$
- ▶ *This is only one possible design for this analysis*