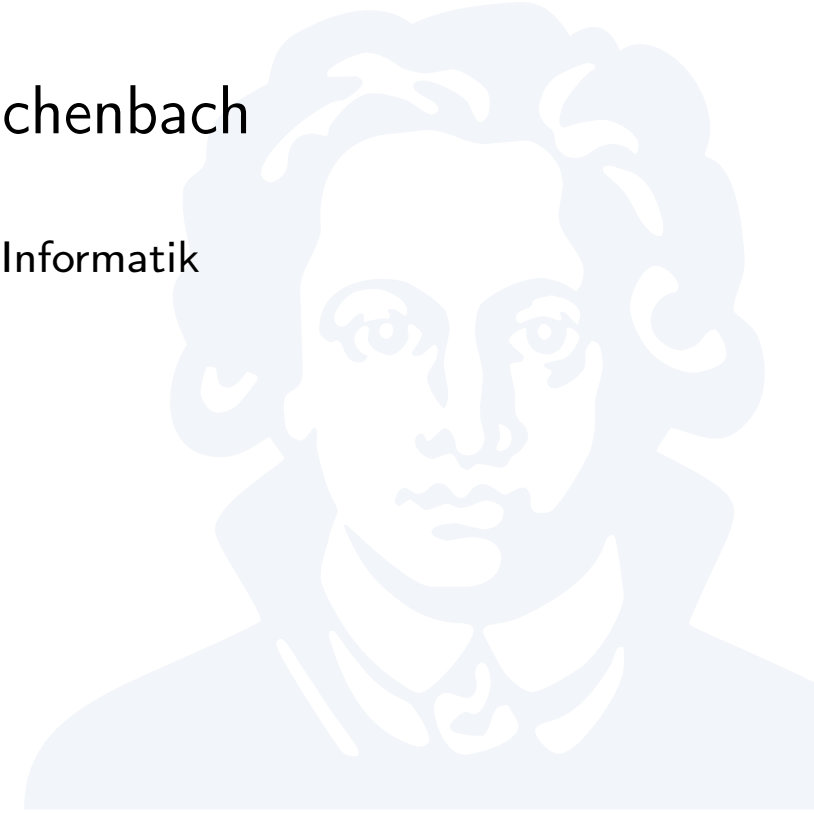
Software Engineering Tools 02
Static Program Analysis

Prof. Dr. Christoph Reichenbach

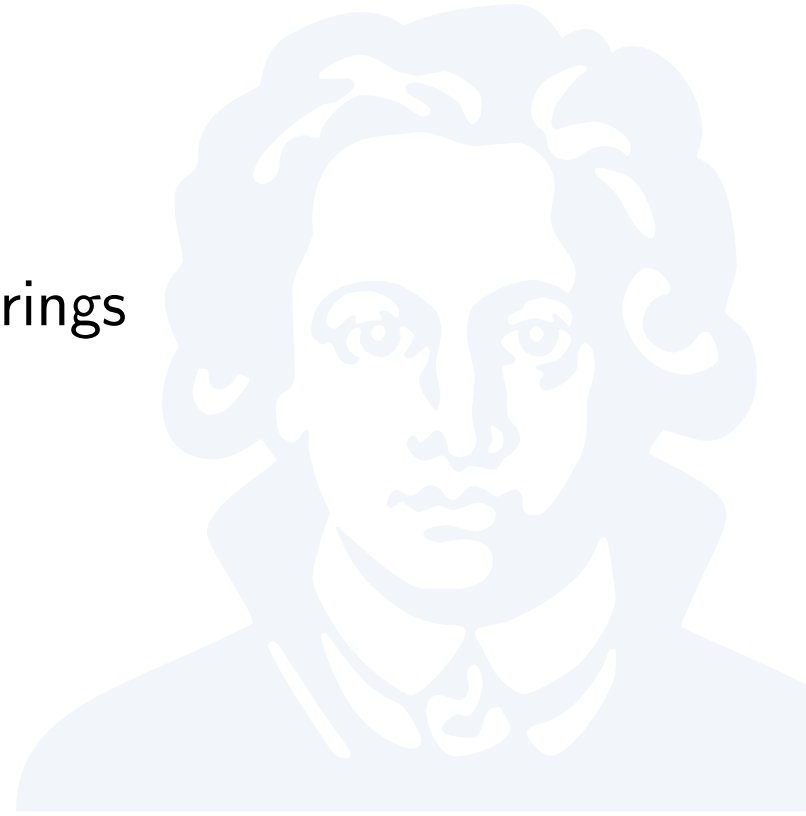
Fachbereich 12 / Institut für Informatik

3. Mai 2014



Why Program Analysis?

- Program understanding
- Finding bugs
- Finding optimisation opportunities
- Checking pre-conditions of refactorings



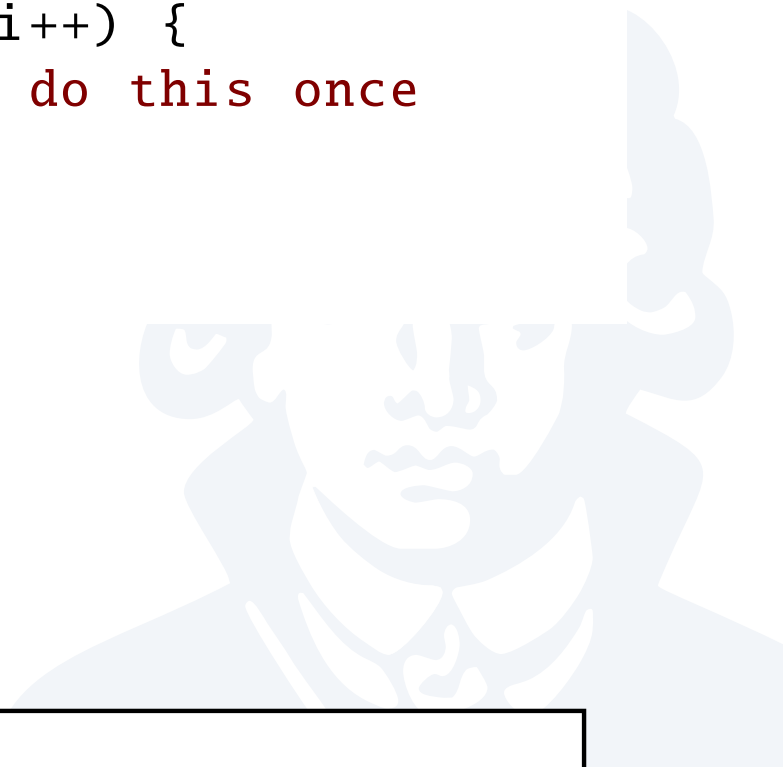
Program analysis for Optimisation

Example: loop-invariant code motion optimisation

- Move code out of loop if this doesn't alter behaviour

```
int global;
void f(int[] a, int[] b)
{
  for (int i = 0; i < a.length; i++) {
    global = 0;  // only need to do this once
    a[i] += b[i];
  }
}
```

- Detect that code is invariant
-
-



Program analysis for Optimisation

Example: loop-invariant code motion optimisation

- Move code out of loop if this doesn't alter behaviour

```
int global;
void f(int[] a, int[] b)
{
  for (int i = 0; i < a.length; i++) {
    a[i] += b[i];
    b[i] = i + Math.atan2(a.length,b.length);
  }
}
```

- Detect that code is invariant
- Invariant code may be *subexpression*
-

Only Math.atan2(...) should be moved

Program analysis for Optimisation

Example: loop-invariant code motion optimisation

- Move code out of loop if this doesn't alter behaviour

```
int global;
void f(int[] a, int[] b)
{
  for (int i = 0; i < a.length; i++) {
    a[i] += b[i];
    global = b[7];
  }
}
```

- Detect that code is invariant
- Invariant code may be *subexpression*
- Must consider *pointer aliasing*

Can't move easily if a = b!

Today's Topics

- **Type and effect systems**
- Intermediate representations
- **Data-flow analysis**
- **Constraint-based analysis**
- **Abstract Interpretation**
- Analysis Qualities
- Other interesting analyses

Type-and-Effect systems

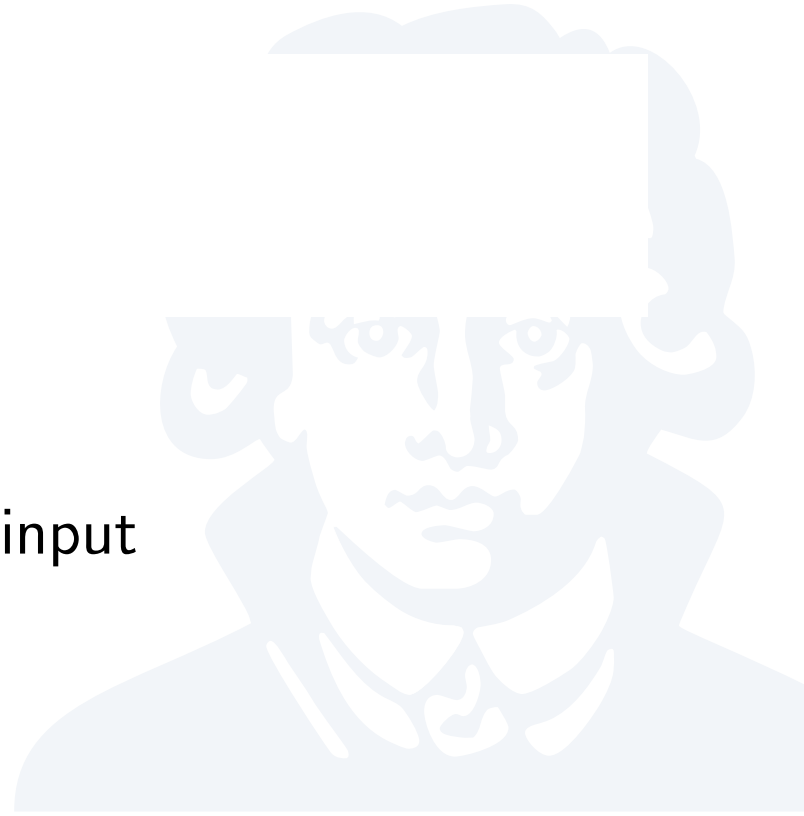
Motivation

- Type analysis can tell us many interesting properties
- How about side effects?

```
(* Example in Standard ML *)

fun f(x) = x*x
fun g(x) = (print "Foo"; x*x)
```

- Types of `f`, `b`: `int -> int`
- Both compute the square of their input
- `g` *also* produces side effect

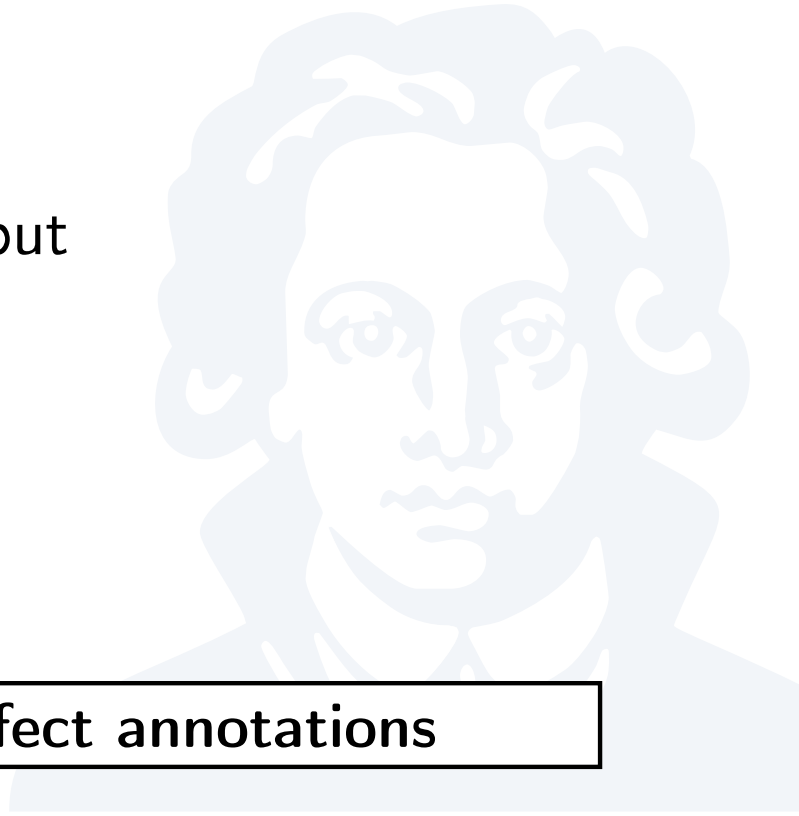


Arrows with effect annotations

Effect annotation

$$\tau_1 \xrightarrow{\eta} \tau_2$$

- η captures side effects
- Example: $\{\texttt{io}\} \in F$ for input/output
- We can now distinguish:
  - $\texttt{f} : \text{int} \to \text{int}$
  - $\texttt{g} : \text{int} \xrightarrow{\{\texttt{io}\}} \text{int}$



Enrich function arrows with effect annotations

Effect environments

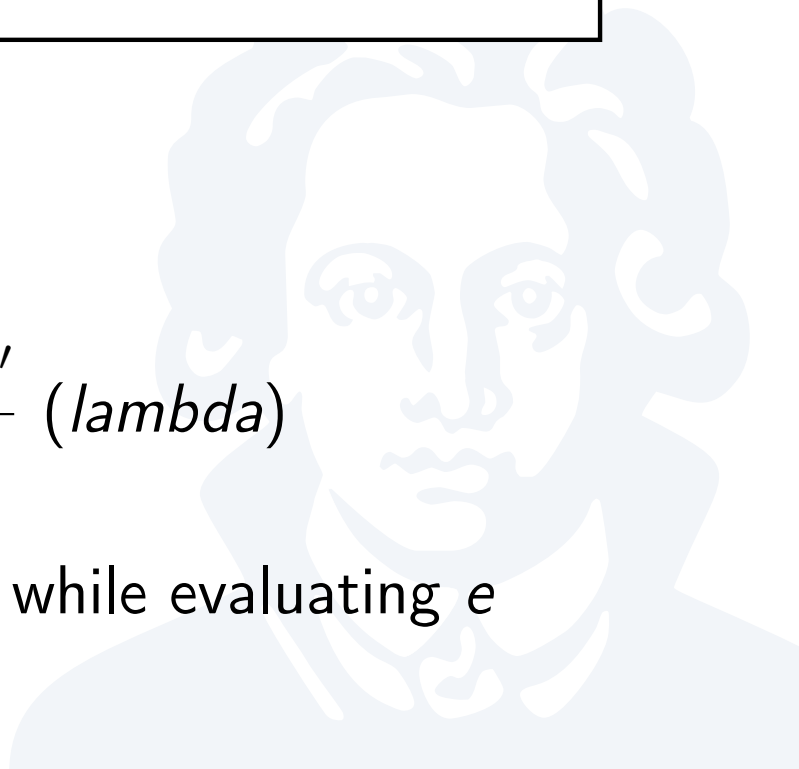
Let F be the set of all effects. We use:

Effect set: $E : 2^F$

- We now type with $\Gamma; E \vdash e$
- Example:

$$\frac{\Gamma + x \mapsto \alpha; E \vdash e : \tau'}{\Gamma; \emptyset \vdash \lambda x.e : \alpha \xrightarrow{E} \tau} \ (lambda)$$

- E captures all effects that happen while evaluating e



Example

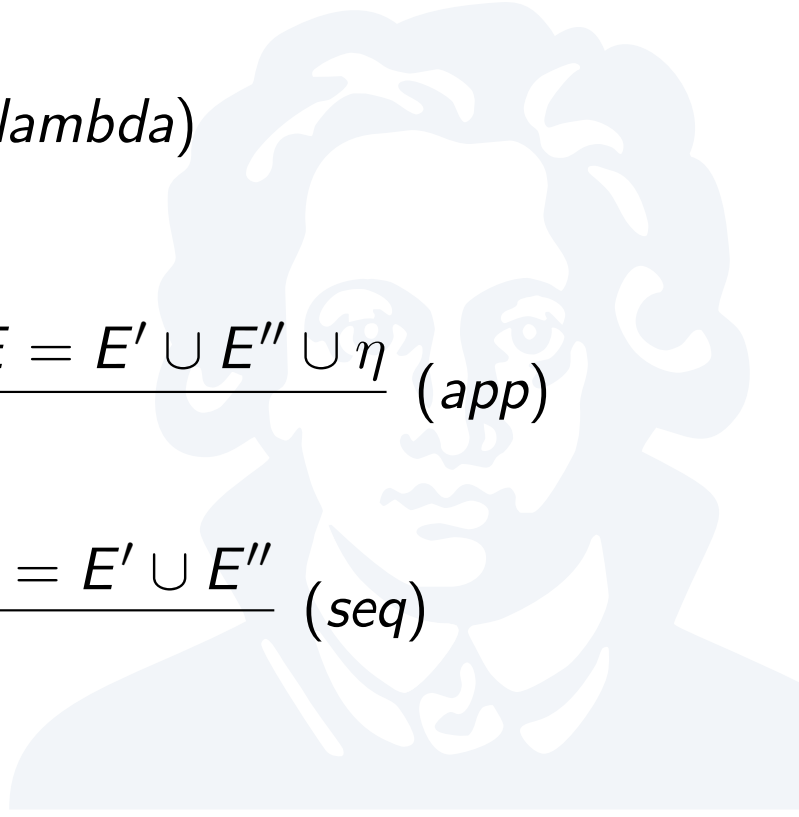
Example for an effect system on a functional language:

$$\frac{\Gamma(n) = \tau}{\Gamma; \emptyset \vdash n : \tau} \ (var)$$

$$\frac{\Gamma + x \mapsto \alpha; E \vdash e : \tau'}{\Gamma; \emptyset \vdash \lambda x.e : \alpha \xrightarrow{E} \tau} \ (lambda)$$

$$\frac{\Gamma; E' \vdash f : \tau_1 \xrightarrow{\eta} \tau_2 \quad \Gamma; E'' \vdash e : \tau_1 \quad E = E' \cup E'' \cup \eta}{\Gamma; E \vdash f \ e : \tau_2} \ (app)$$

$$\frac{\Gamma; E' \vdash e_1 : \alpha \quad \Gamma; E'' \vdash e_2 : \tau \quad E = E' \cup E''}{\Gamma; E \vdash (e_1 ; e_2) : \tau} \ (seq)$$



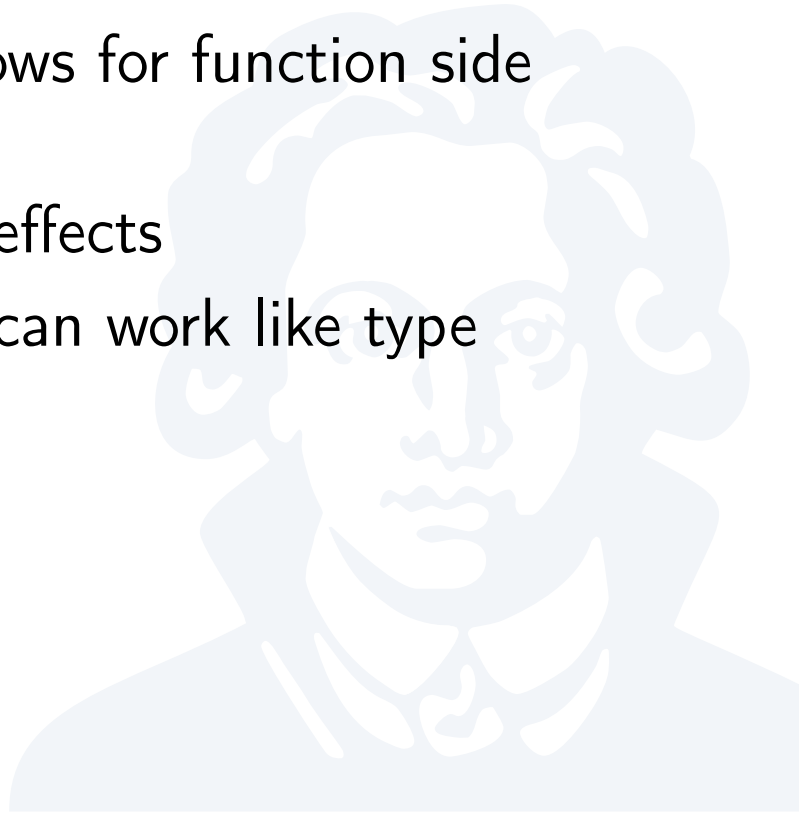
Analysis

$$\dfrac{\dfrac{\dfrac{\Gamma(\text{print}) = \text{string} \xrightarrow{\{\texttt{io}\}} \text{unit}}{\Gamma'; \emptyset \vdash \text{print} : \text{string} \xrightarrow{\{\texttt{io}\}} \text{unit}} \quad \ldots}{\Gamma'; \{\texttt{io}\} \vdash \text{print } \text{"..."} : \text{unit}} \quad \dfrac{\Gamma'(x) = \text{int}}{\Gamma'; \emptyset \vdash x : \text{int}} \quad \{\texttt{io}\} = \{\texttt{io}\} \cup \emptyset}{\Gamma'; \{\texttt{io}\} \vdash (\text{print } \text{"..."}; x) : \text{int}}}{\Gamma; \emptyset \vdash (\lambda x.(\text{print } \text{"..."}; x)) : \text{int} \xrightarrow{\{\texttt{io}\}} \text{int}}$$

where $\Gamma' = \Gamma + x \mapsto \text{int}$

Summary: Type-and-Effect Systems

- Extension of type systems
- *effect annotations* on function arrows for function side effects
- Effect environments for local side effects
- Not covered here: *effect variables* can work like type variables



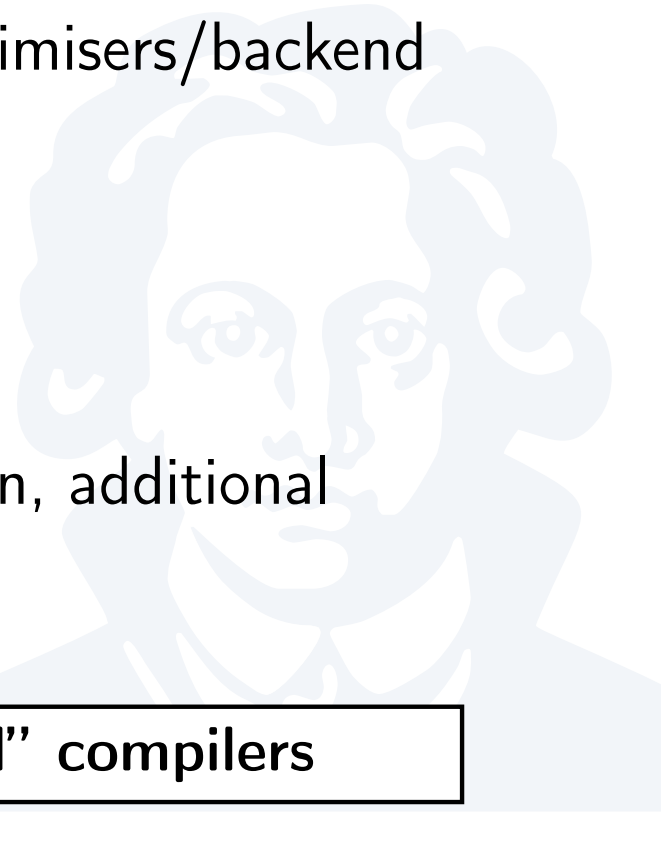
Intermediate Representation

Intermediate representations (IRs) simplify compiler construction:

- Encode knowledge needed for many analyses
- Allow compiler backend, optimisers, frontend to evolve separately
- Allow simplifying the language that optimisers/backend operate on
  $\Rightarrow$ faster evolution, fewer bugs

Disadvantages:

- Require AST $\rightarrow$ IR translation
- For *non-optimising* compilers: slow-down, additional complexity



IRs used in practically all "real-world" compilers

Basic Blocks

```
int x = 0;
int y = a;
if (y > x) {
  x = 1;
} else {
  x = 2;
  y = 0;
}
y = x + y;
return y;
```

```
int x = 0;
int y = a;
if (y > x)
```

```
x = 1;
```

```
x = 2;
y = 0;
```

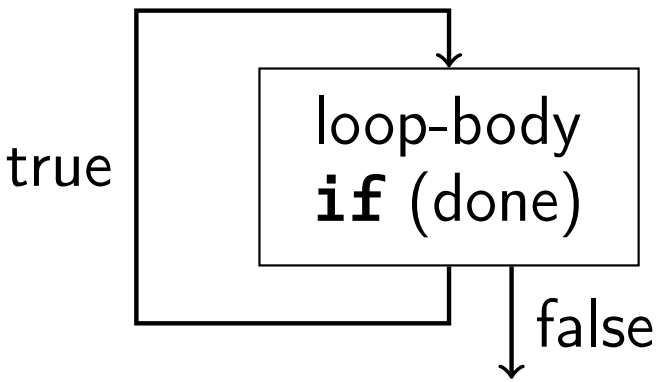
```
y = x + y;
return y;
```

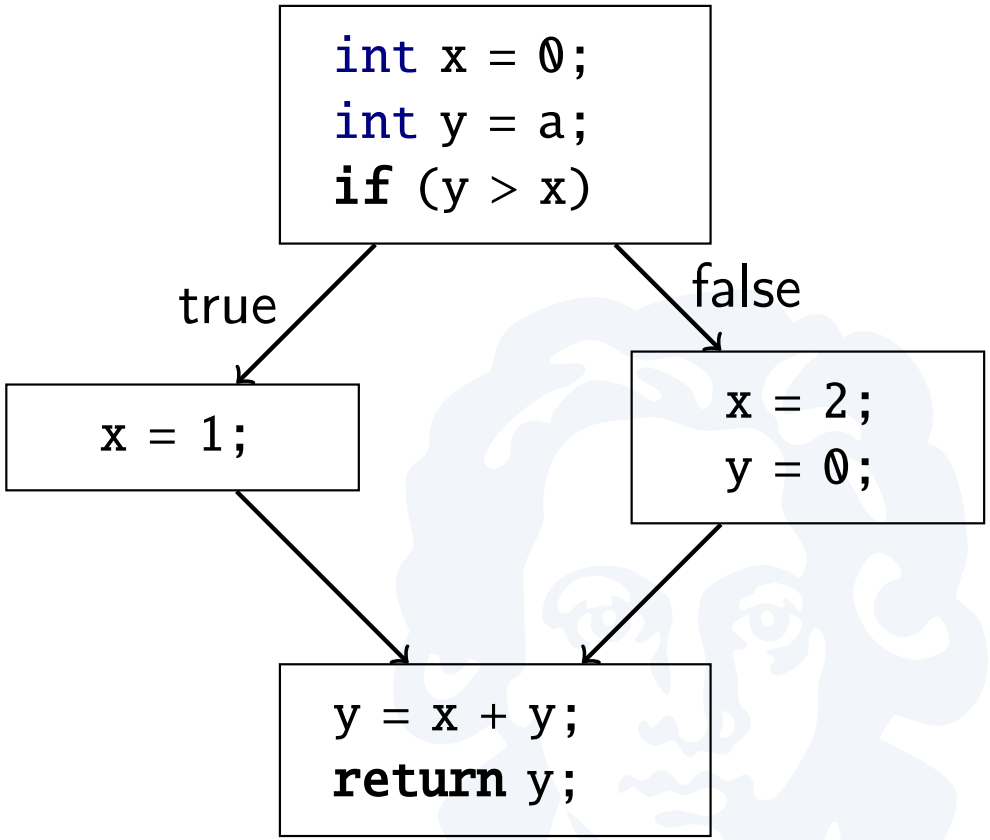
Group operations without intervening control flow

Control-Flow Graphs

Maintain *control edges*

- Labelled: '*true*'/`false` for conditional jumps
- Unlabelled: unconditional jump
- Loops: Back-edges



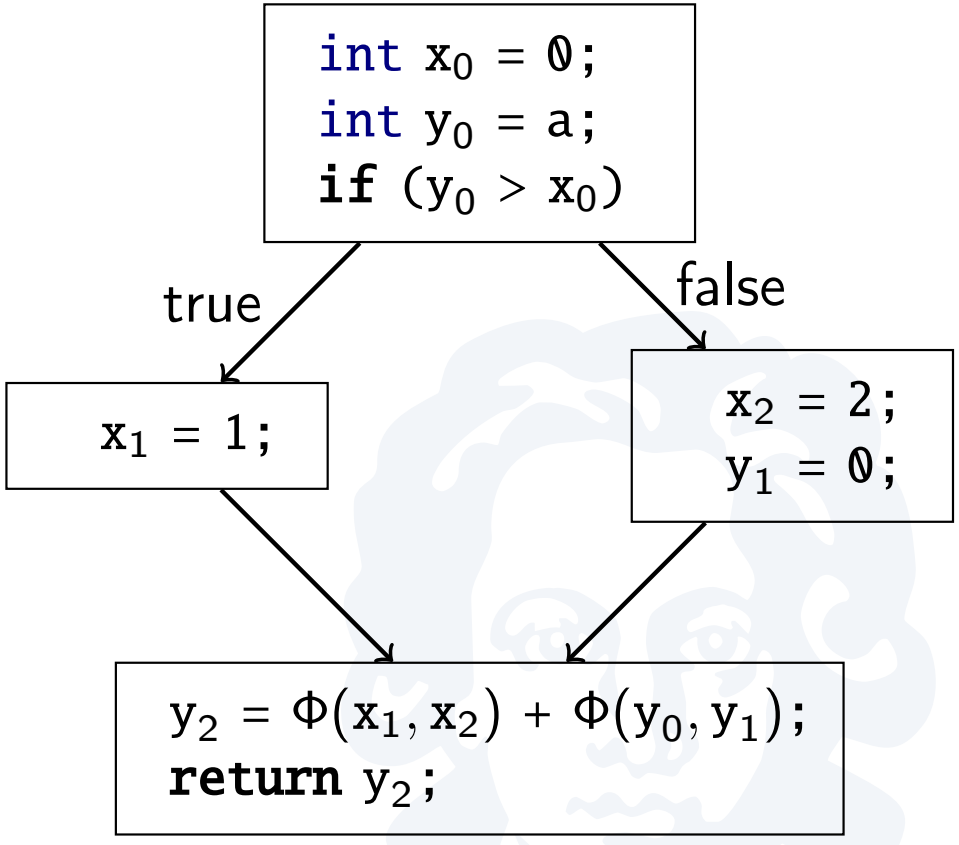


Needed in order for basic blocks to make sense

Static Single-Assignment Form (SSA)

Assign each variable exactly once

- Rename variables to ensure this property
- When different variables flow together, mark with $\Phi(v_1, \ldots, v_k)$
- Φ makes dependencies explicit, has trivial semantics (exactly one v_i will be set)

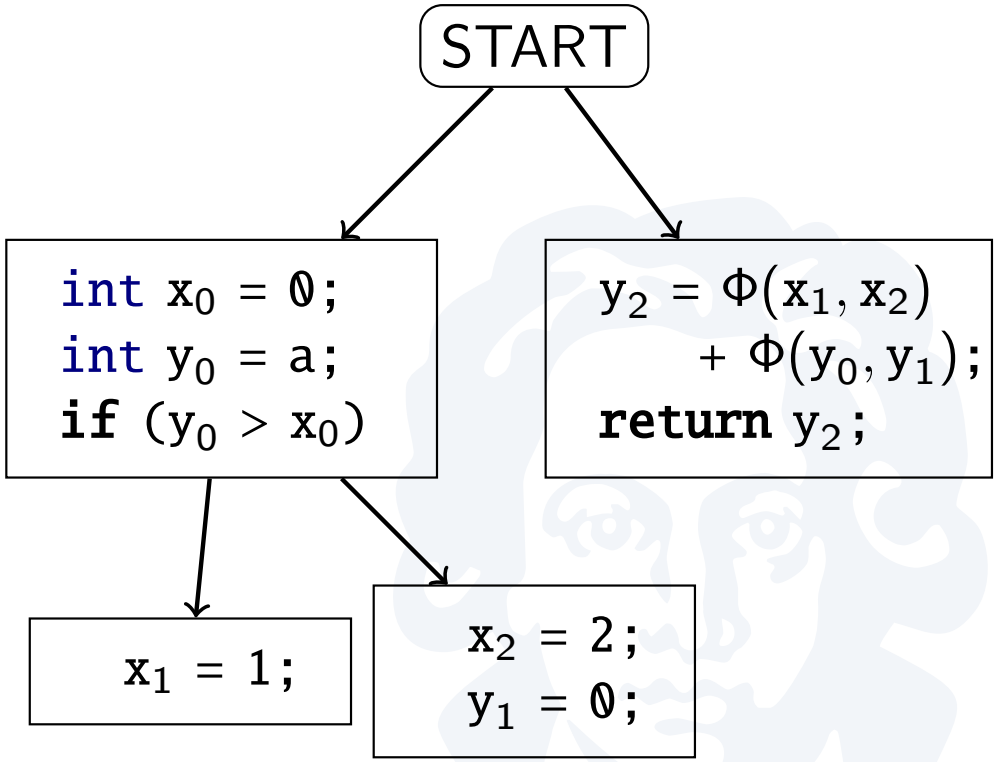


Simplifies or enables many analyses

Control-Dependence Graph

Capture nontrivial dependencies between basic blocks

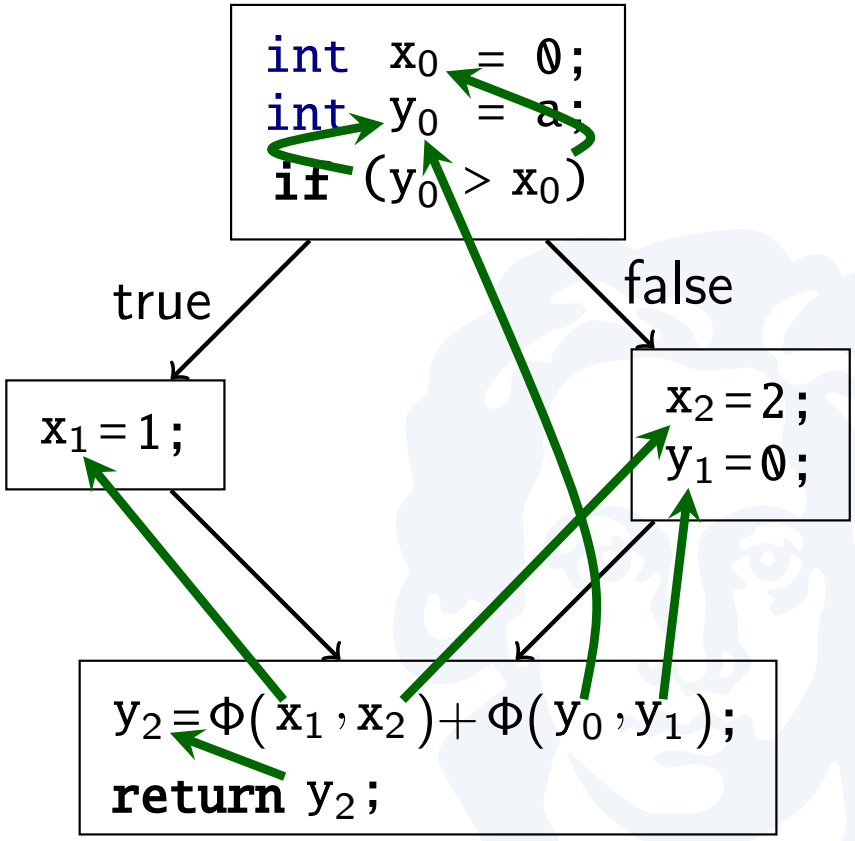
- Arrow $a \longrightarrow b$: Block a decides whether b is invoked



Data-Dependence Graph

Capture dependencies between reads and assignments

- $a \longrightarrow b$: b stores a value that a reads

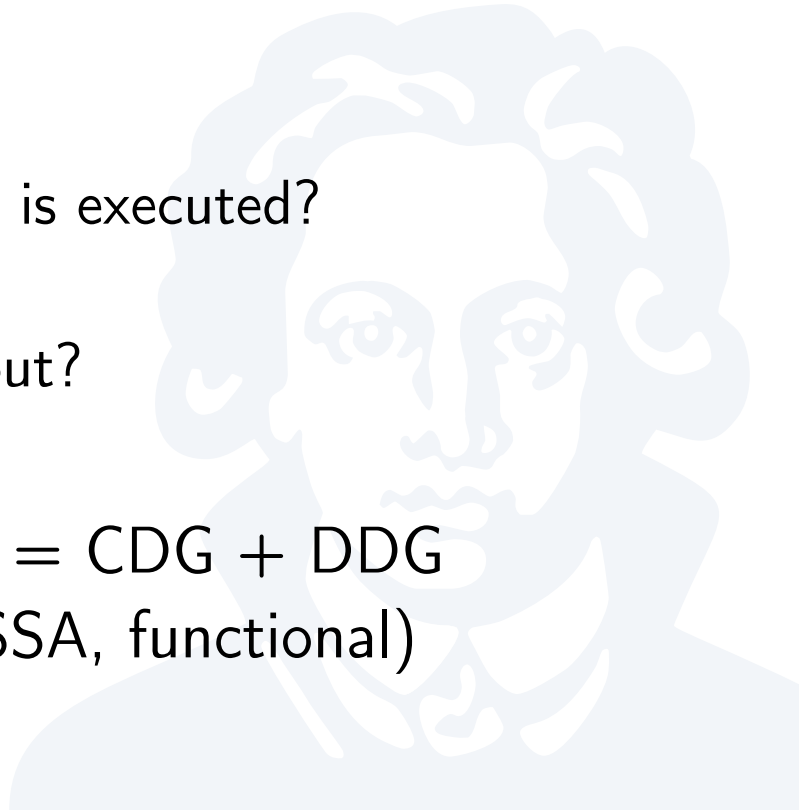


Summary: Intermediate Representations

- Basic Blocks
  - Group operations that always happen together
- SSA (Static Single-Assignment Form)
  - Bind each name exactly once
  - Φ nodes merge multiple inputs
- CDG (Control-Dependence Graph)
  - Who decides whether something is executed?
- DDG (Data-Dependence Graph)
  - Who assigns variables I care about?

Other IRs:

- Program dependence Graph, PDG = CDG + DDG
- A-Normal Form, ANF (similar to SSA, functional)
- Continuation-Passing Form, CPS
  (ANF plus jumps, exceptions, . . .)

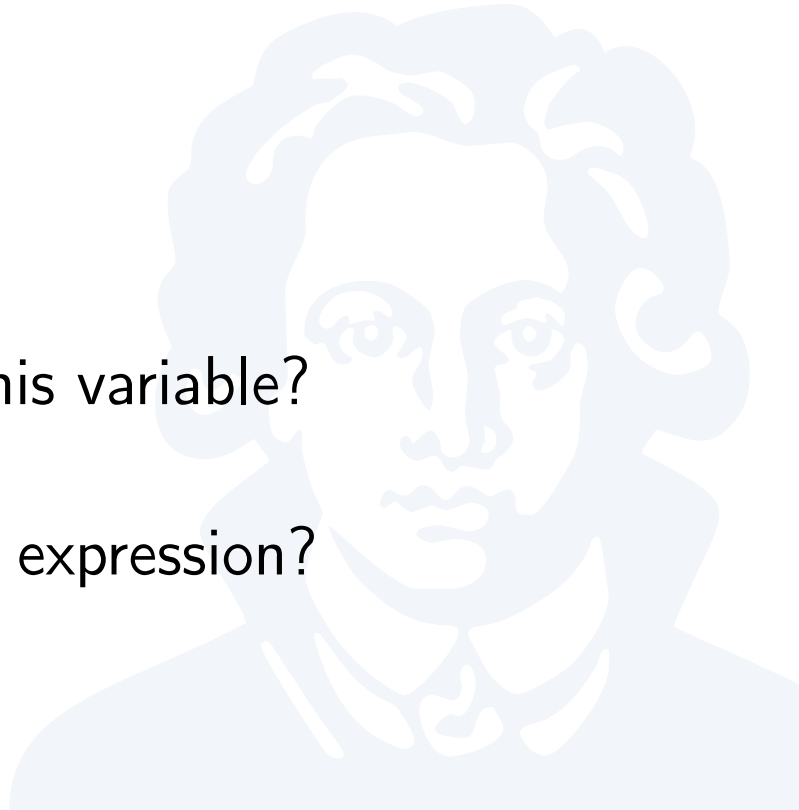


Dataflow Analysis

Analyse properties of variables or basic blocks

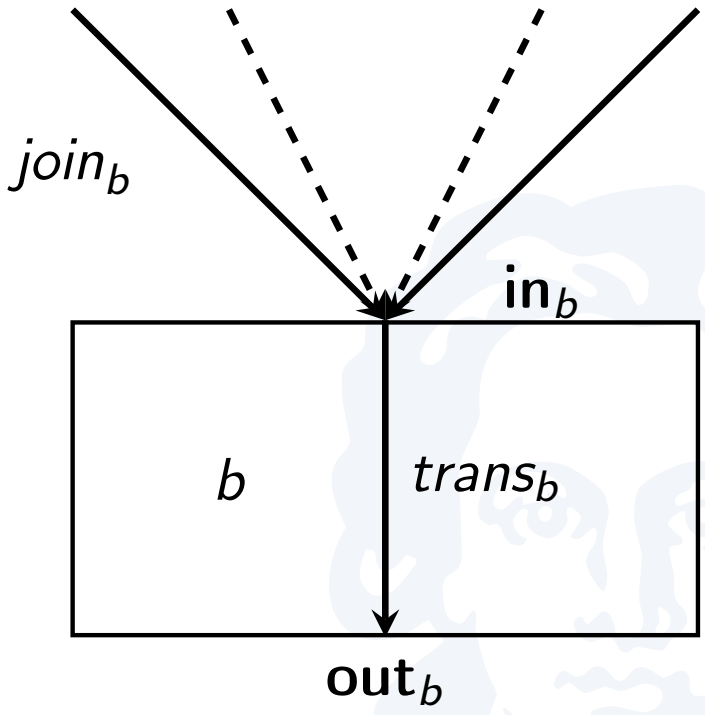
Examples in practice:

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

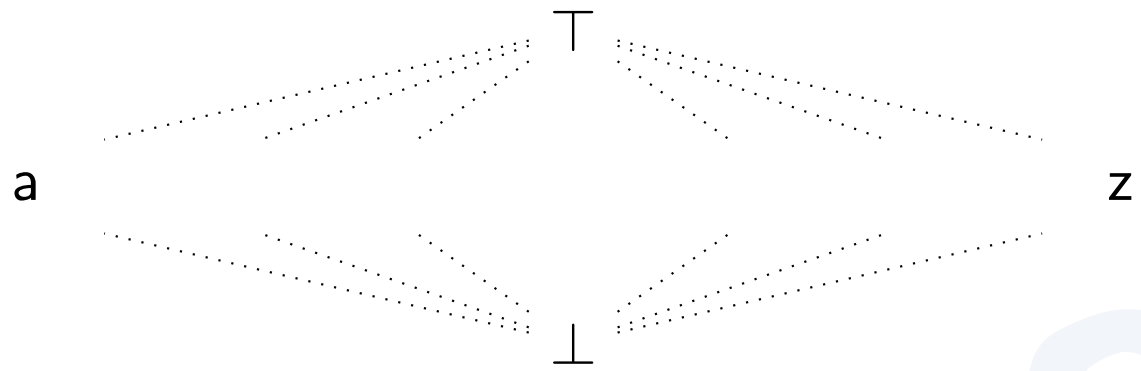


Formalisation

- $\mathbf{in}_b$: knowledge at entrance of basic block b
- $\mathbf{out}_b$: knowledge at exit of basic block b
- $join_b$: merges all $\mathbf{out}_{b_i}$ for all basic blocks b_i that flow into b
- $trans_b$: updates $\mathbf{out}_b$ from $\mathbf{in}_b$



Analysis lattice



- Analysis takes place in a *complete (semi)lattice*
- $\top$: 'Too much information / could be anything'
- $\bot$: 'No information'
- Join functions are typically join or meet operations
- Transfer functions $trans_b$ are *monotonic* (order-preserving)

$$\text{if } a \leq c, \text{ then } t(a) \leq trans_b(c)$$

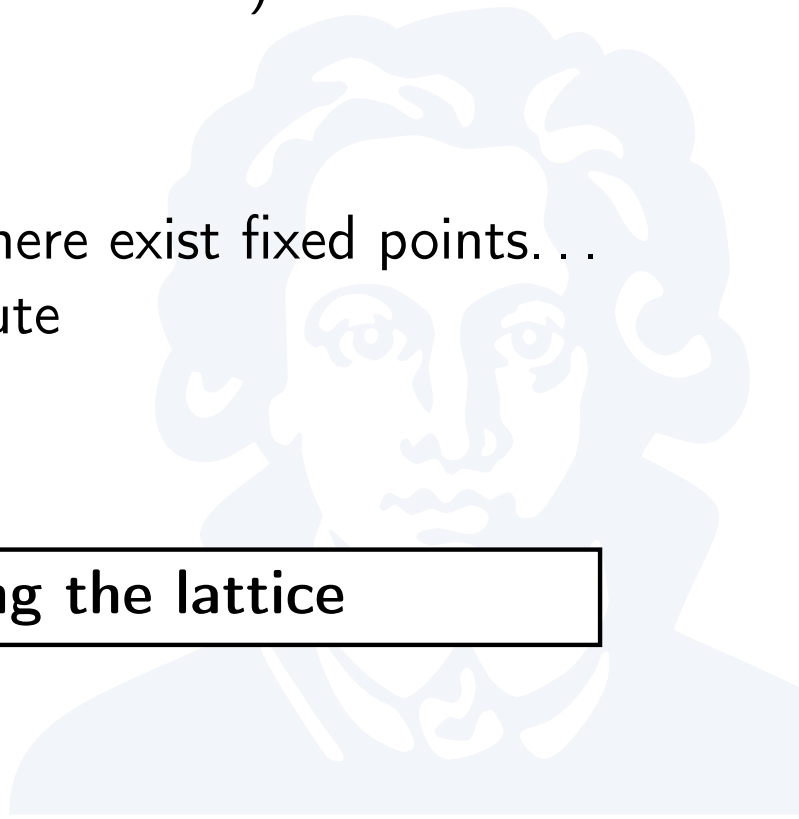
Computing the Least Fixed Point

Computing the results:

- Iterate updating
  - $\mathbf{in}_b := \mathit{join}_b(\mathbf{in}_{b_1}, \ldots, \mathbf{in}_{b_n})$ (b_i flow into b)
  - $\mathbf{out}_b := \mathit{trans}_b(\mathbf{in}_b)$
- Until nothing changes

Knaster-Tarski theorem guarantees that there exist fixed points...
...but these may be impractical to compute

Take care when constructing the lattice

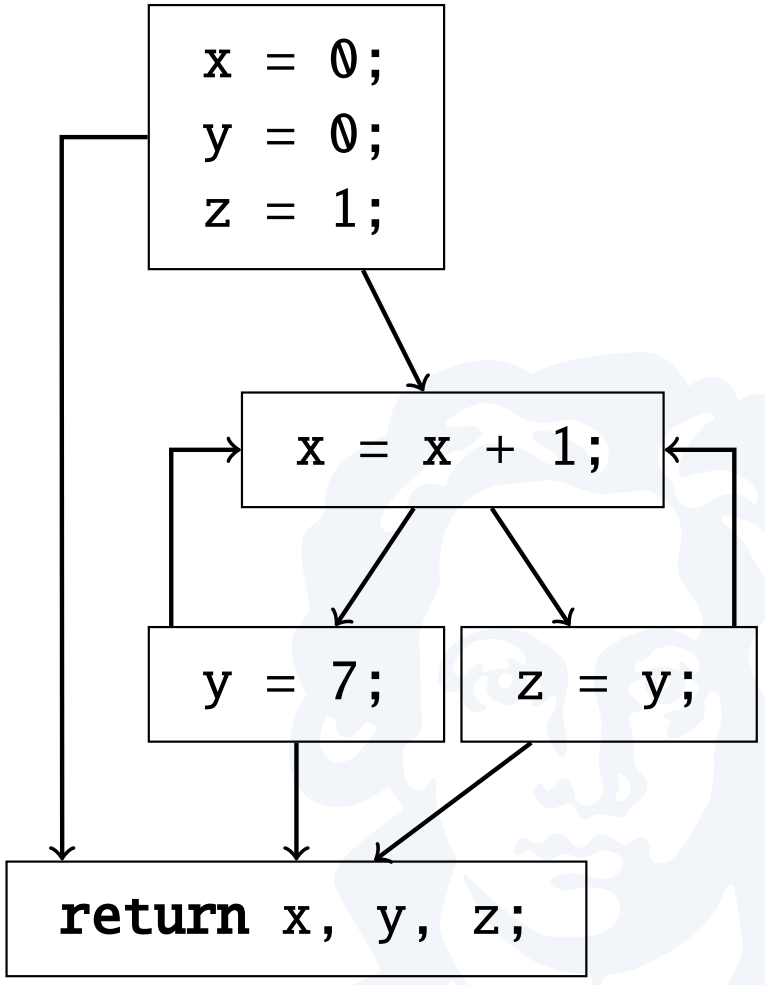


Example: Reaching Definitions

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

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

return x, y, z;
```

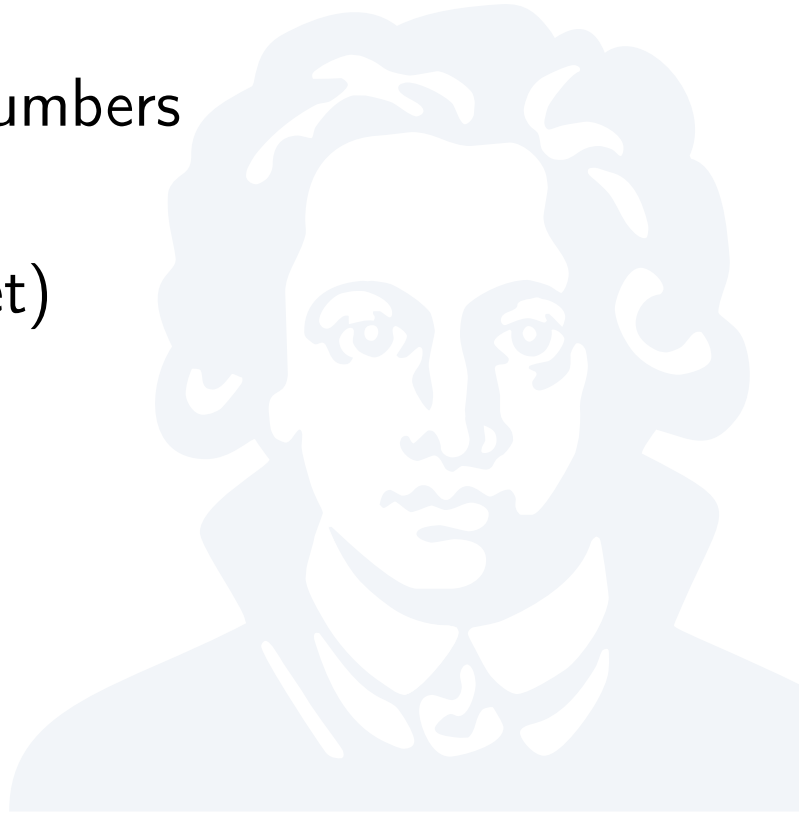


Reaching Definitions: What values are possible?

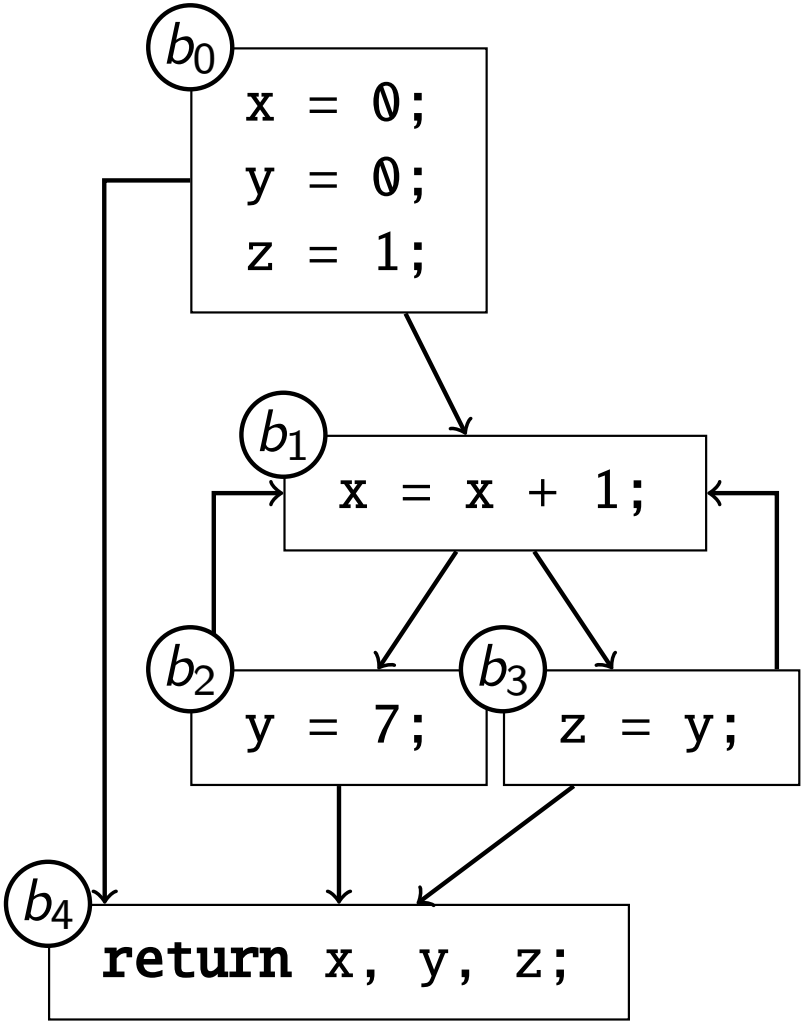
Example: Reaching Definitions

Designing our lattice:

- Capture sets of up to 3 possible numbers
- $\top$: More than 3 possible numbers
- $\bot$: $\emptyset$ (no possible numbers seen yet)



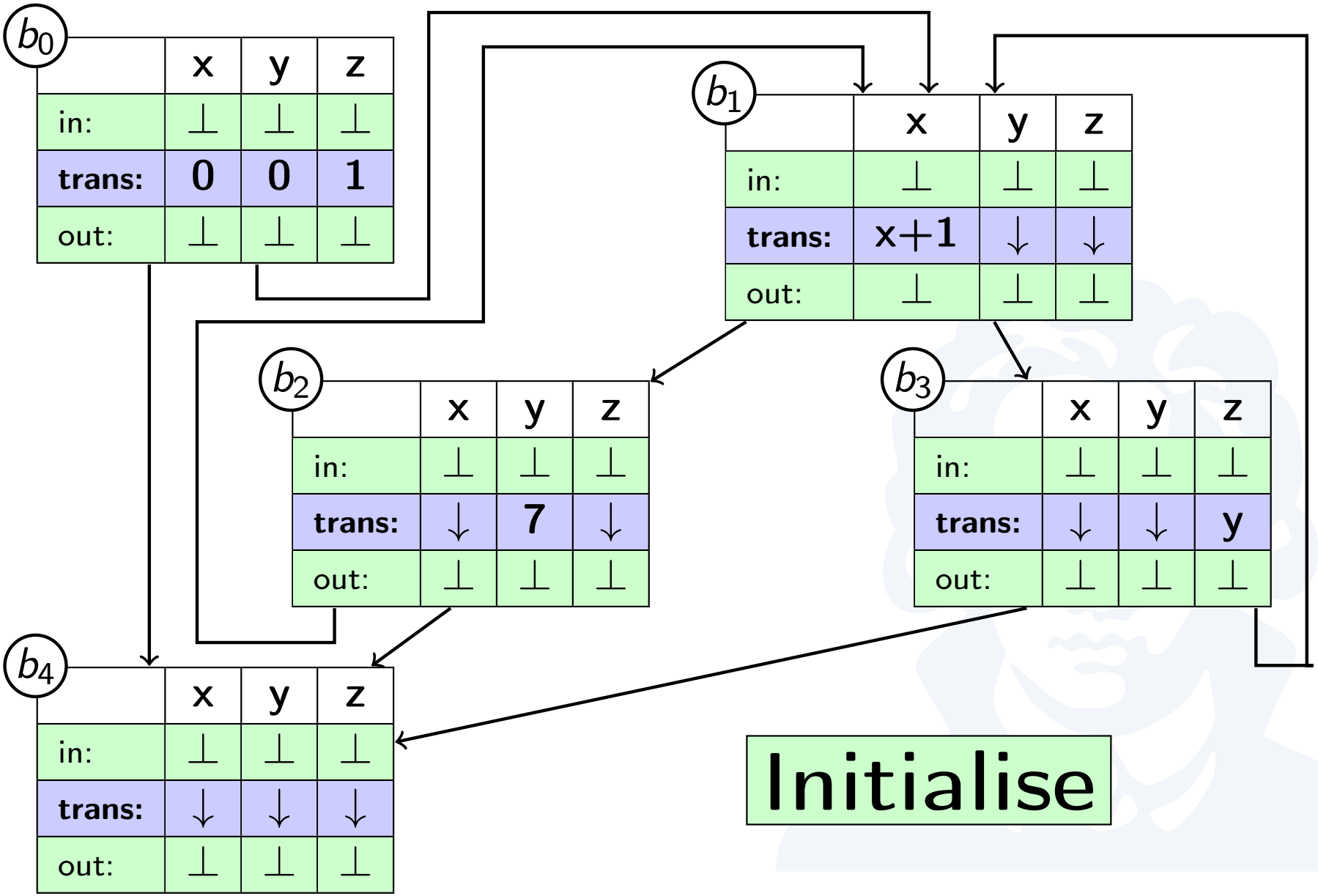
Example: Control-Flow Graph



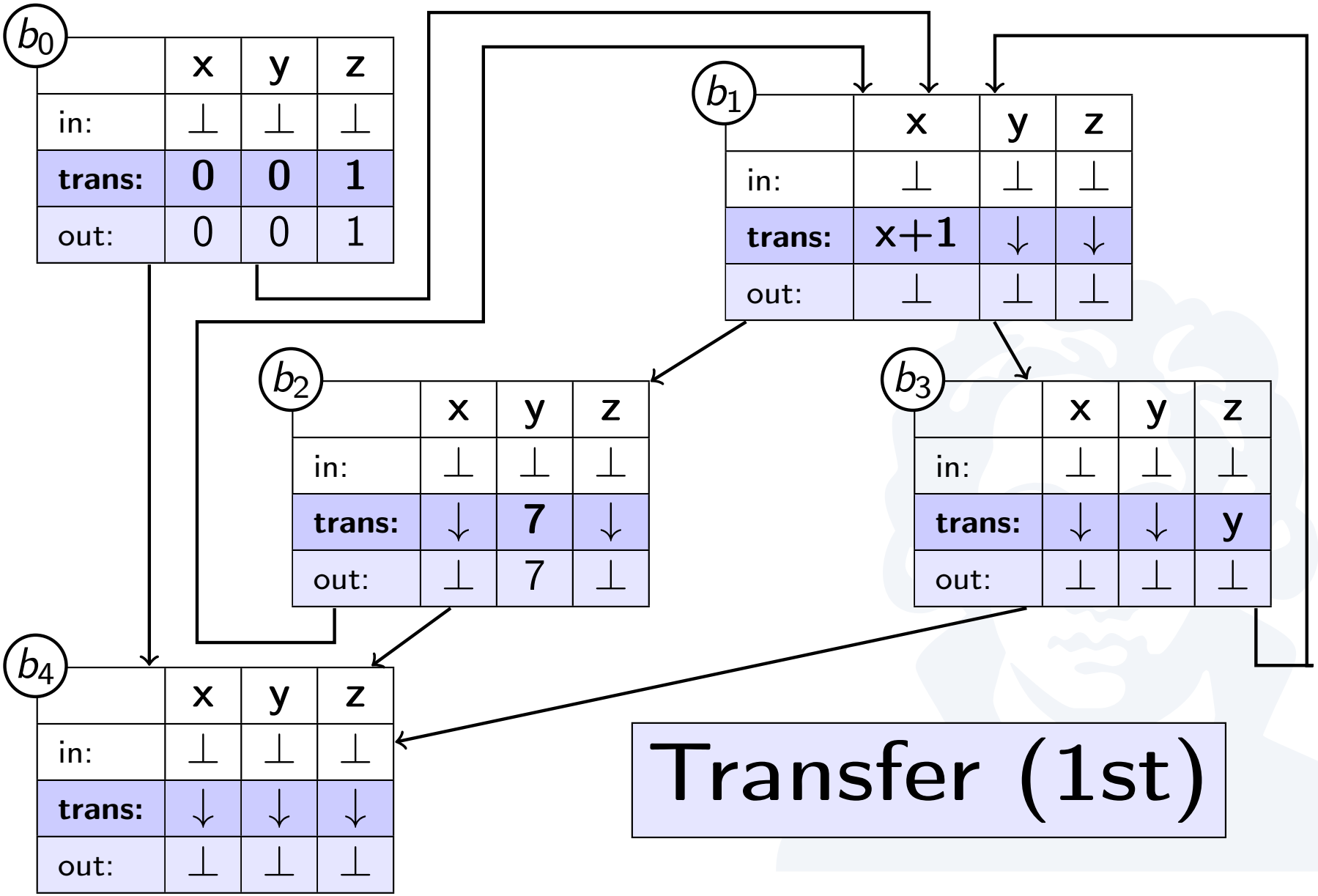
b	inputs	$trans_b$		
		x	y	z
b_0	$\emptyset$	0	0	1
b_1	$\{b_0, b_2, b_3\}$	$x+1$	y	z
b_2	$\{b_1\}$	x	7	z
b_3	$\{b_1\}$	x	y	y
b_4	$\{b_0, b_2, b_3\}$	x	y	z

$$join_b = \bigcup_{s \in \text{inputs}_b} s$$

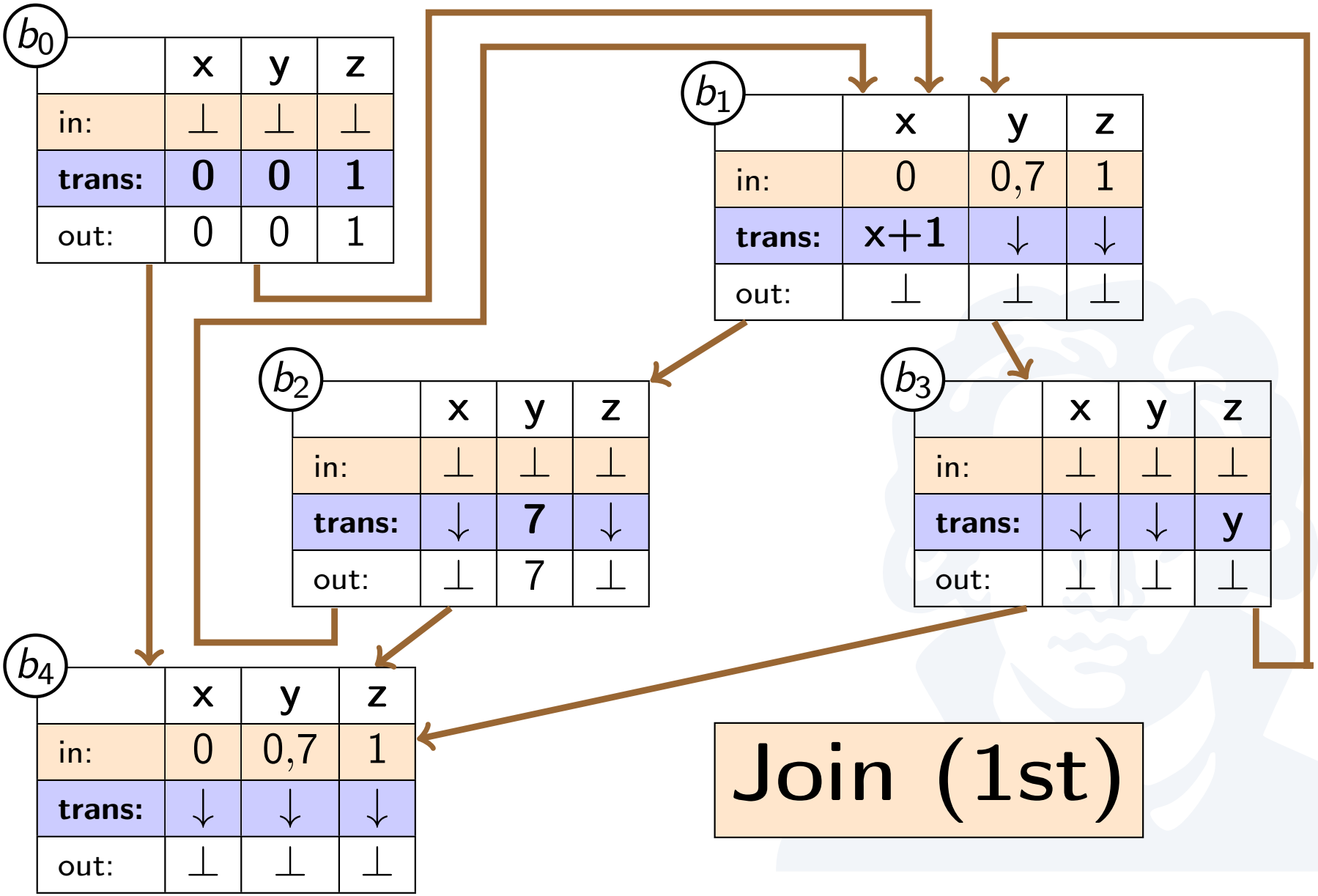
Example: Computing the Fixpoint



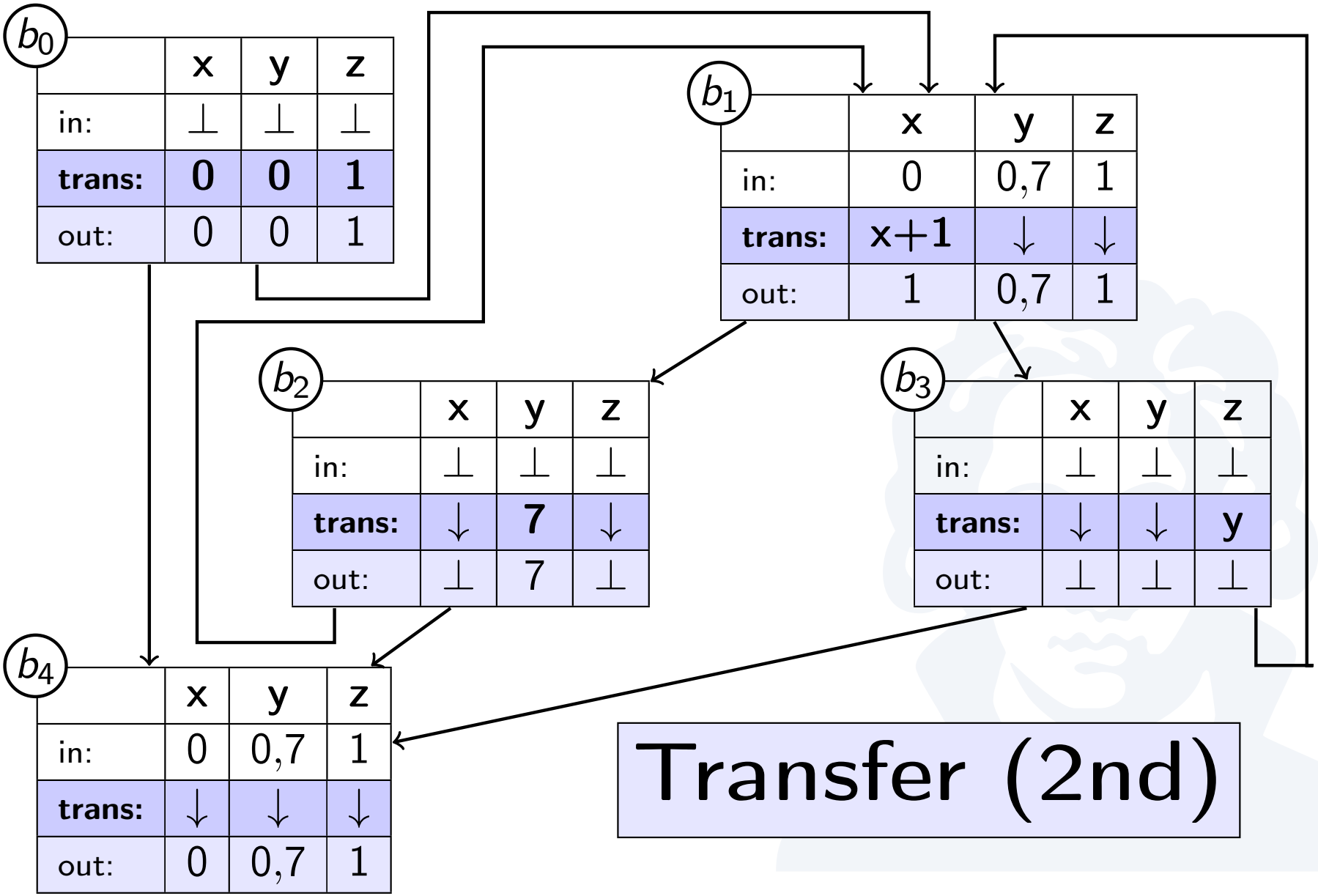
Example: Computing the Fixpoint



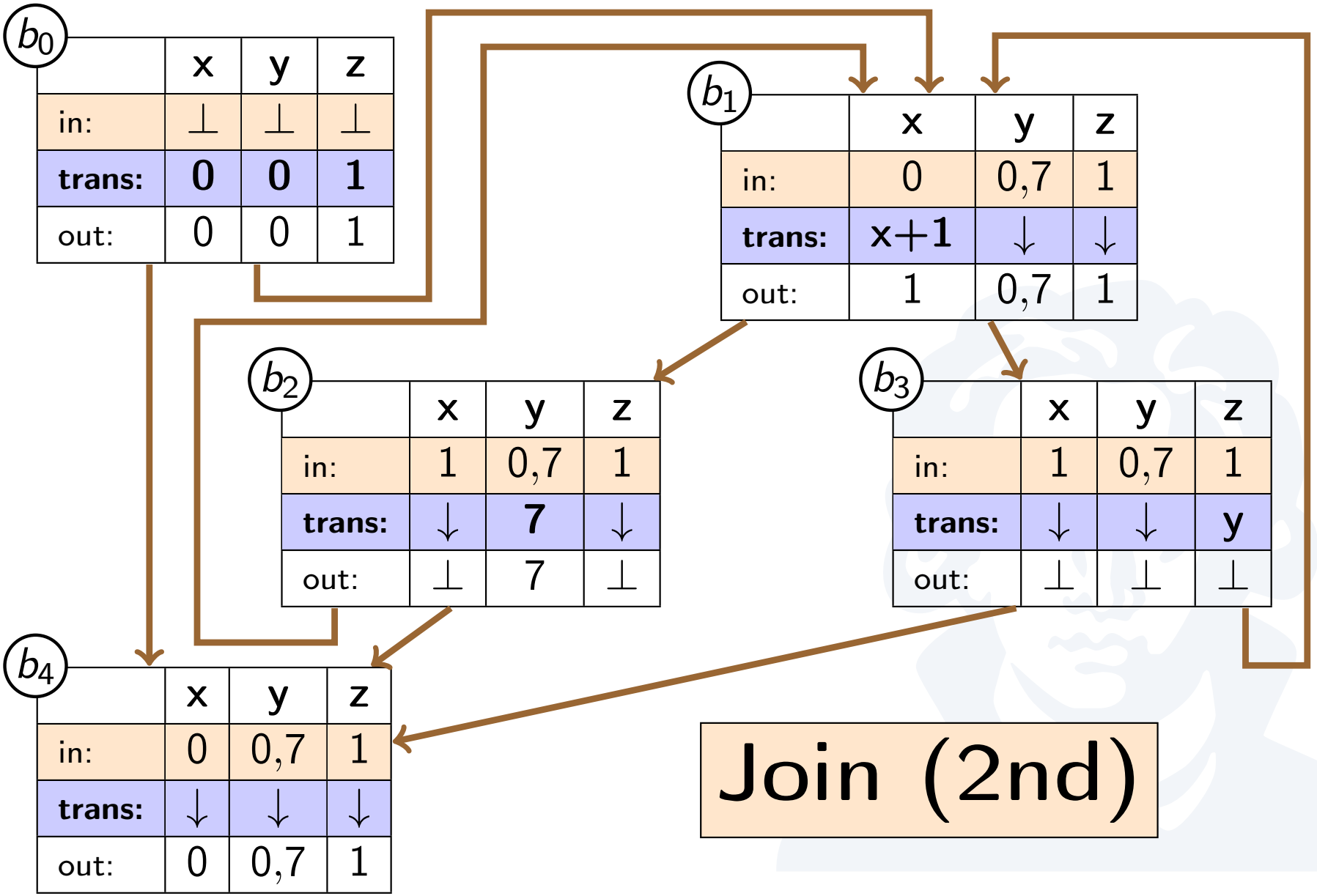
Example: Computing the Fixpoint



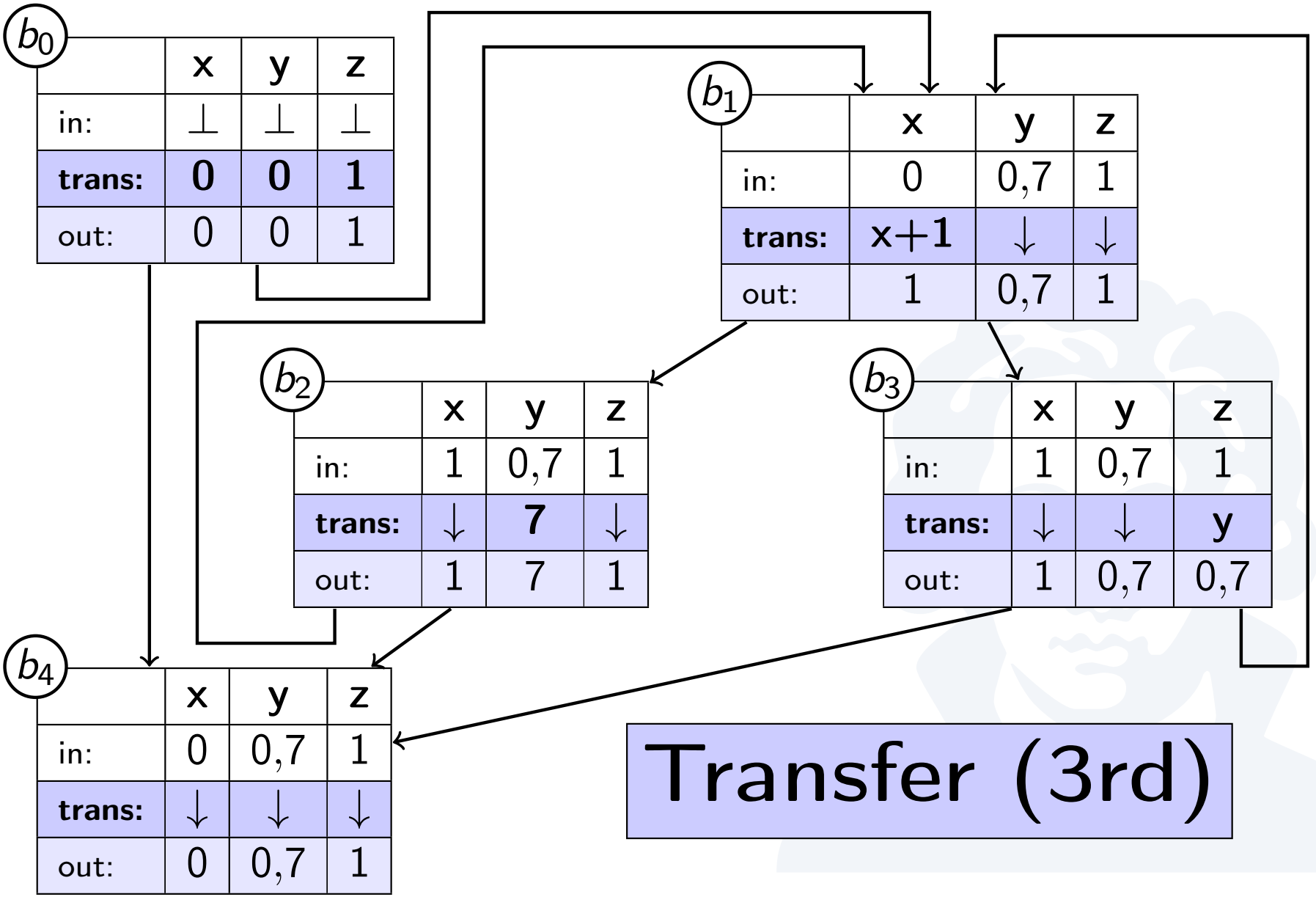
Example: Computing the Fixpoint



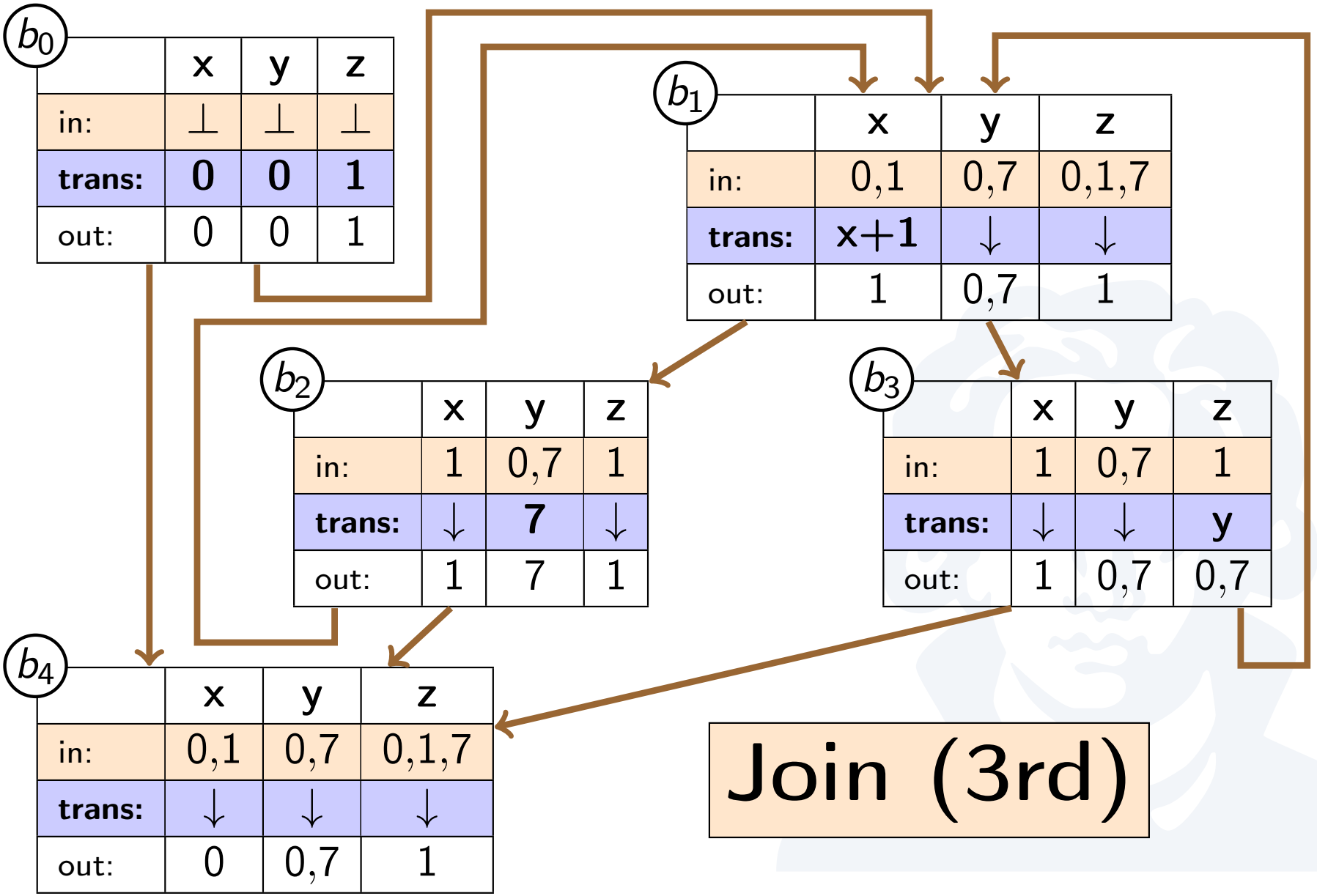
Example: Computing the Fixpoint



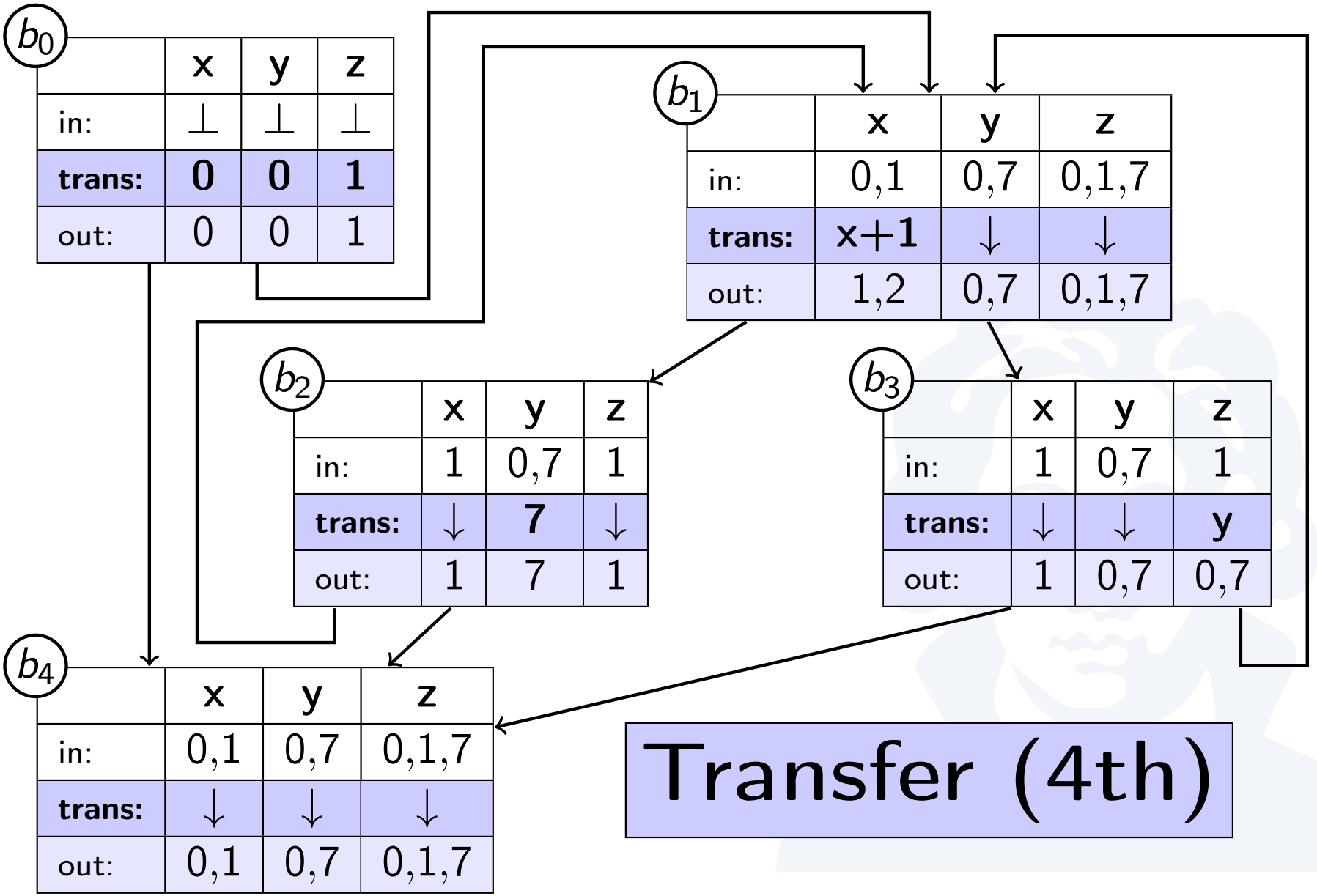
Example: Computing the Fixpoint



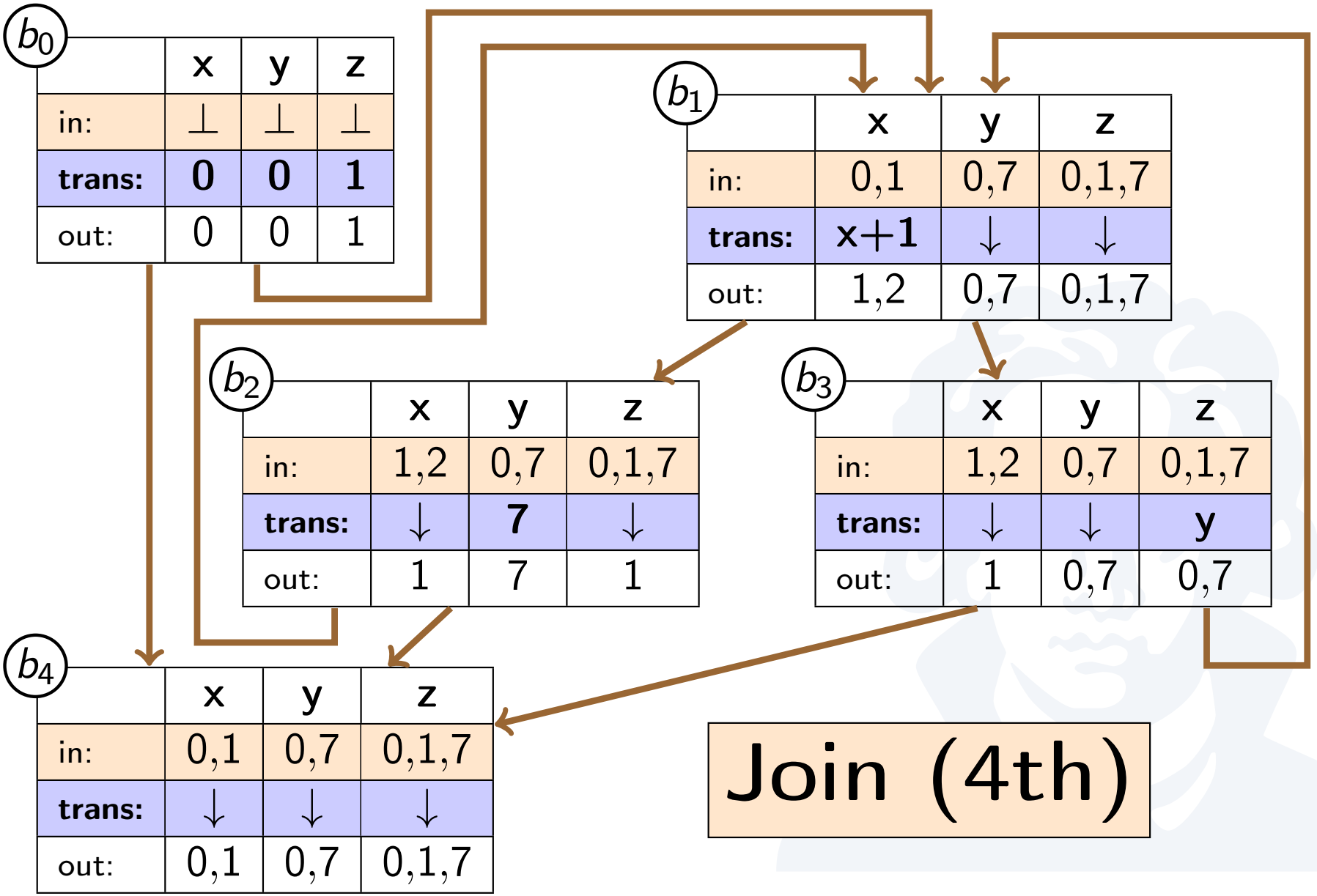
Example: Computing the Fixpoint



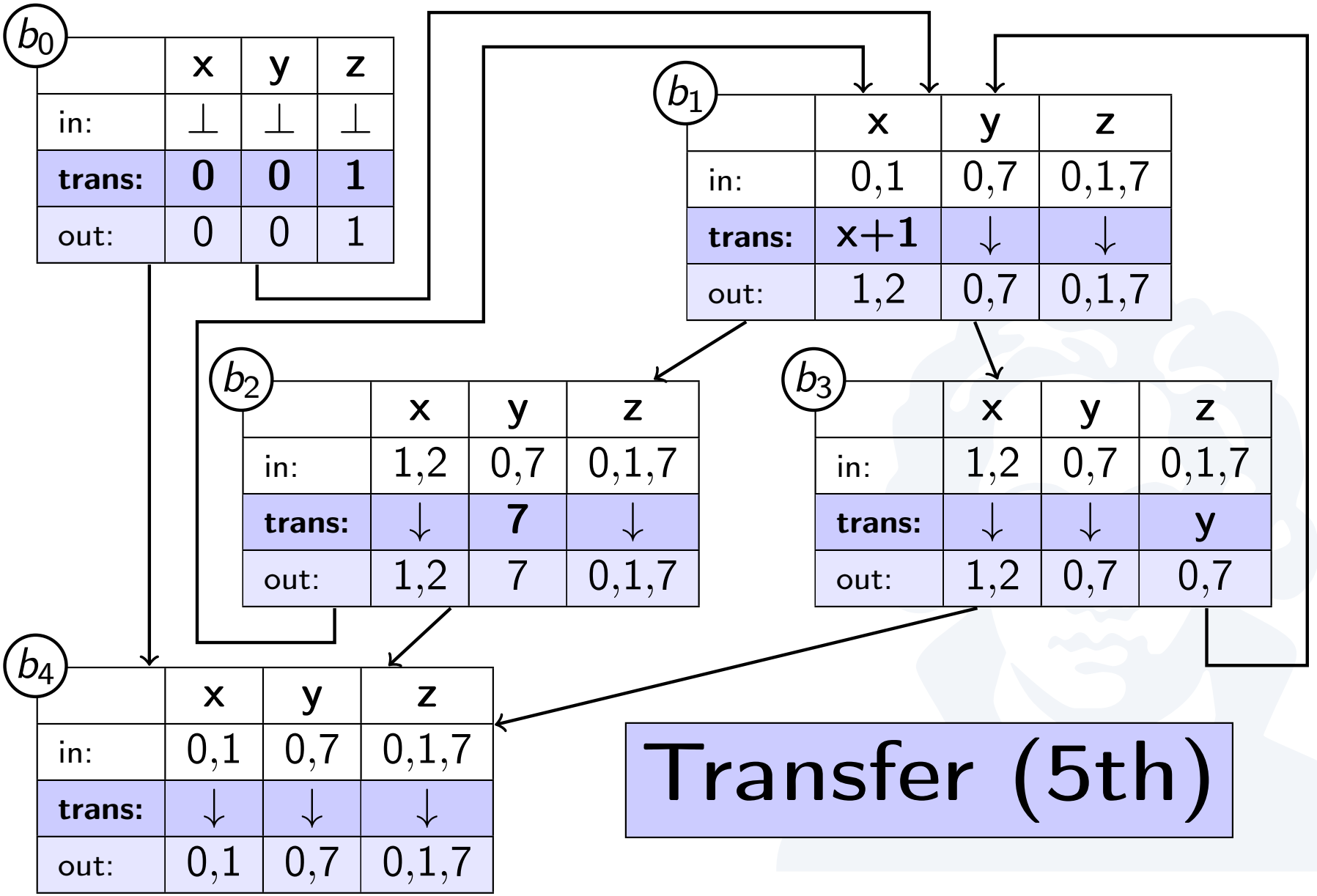
Example: Computing the Fixpoint



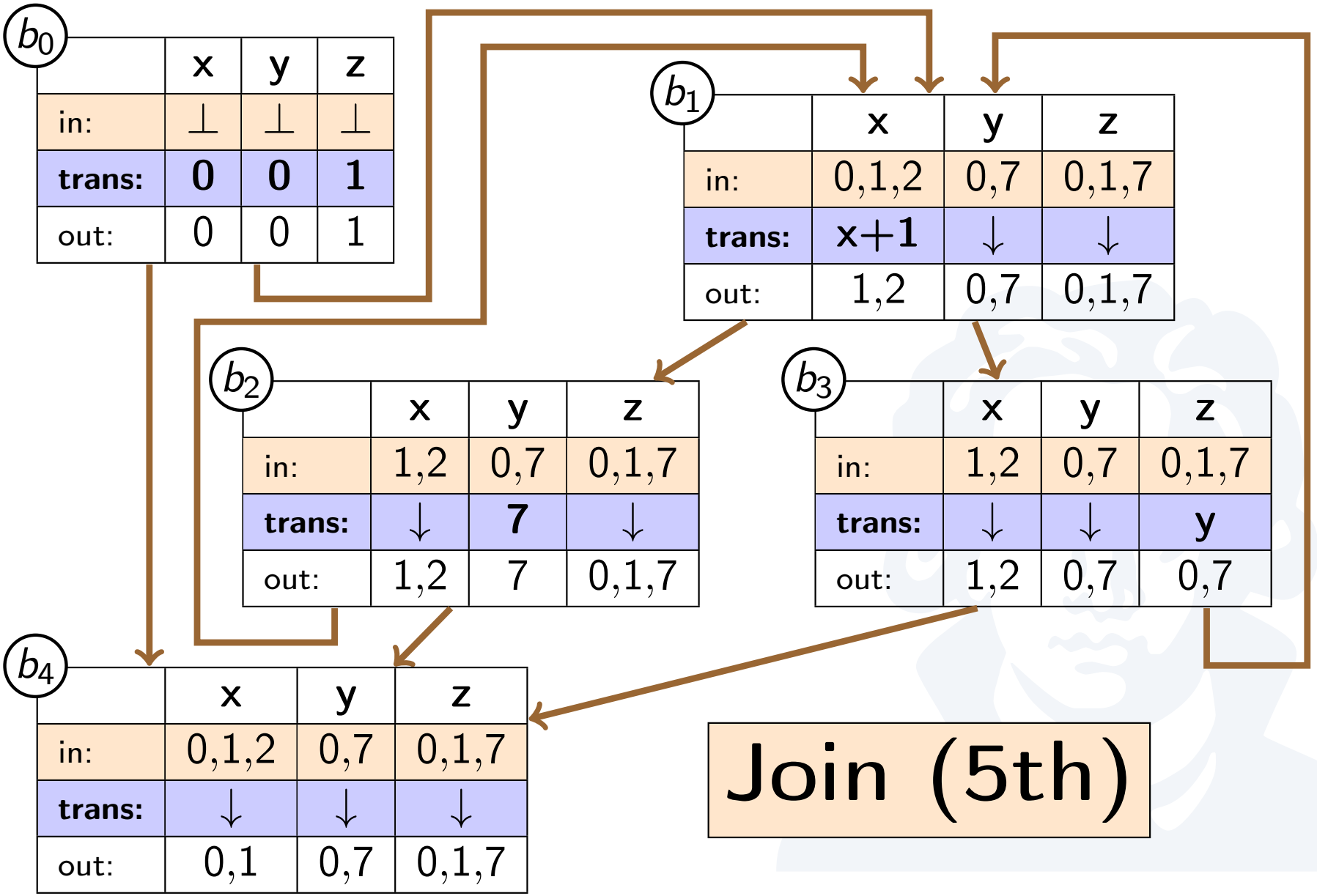
Example: Computing the Fixpoint



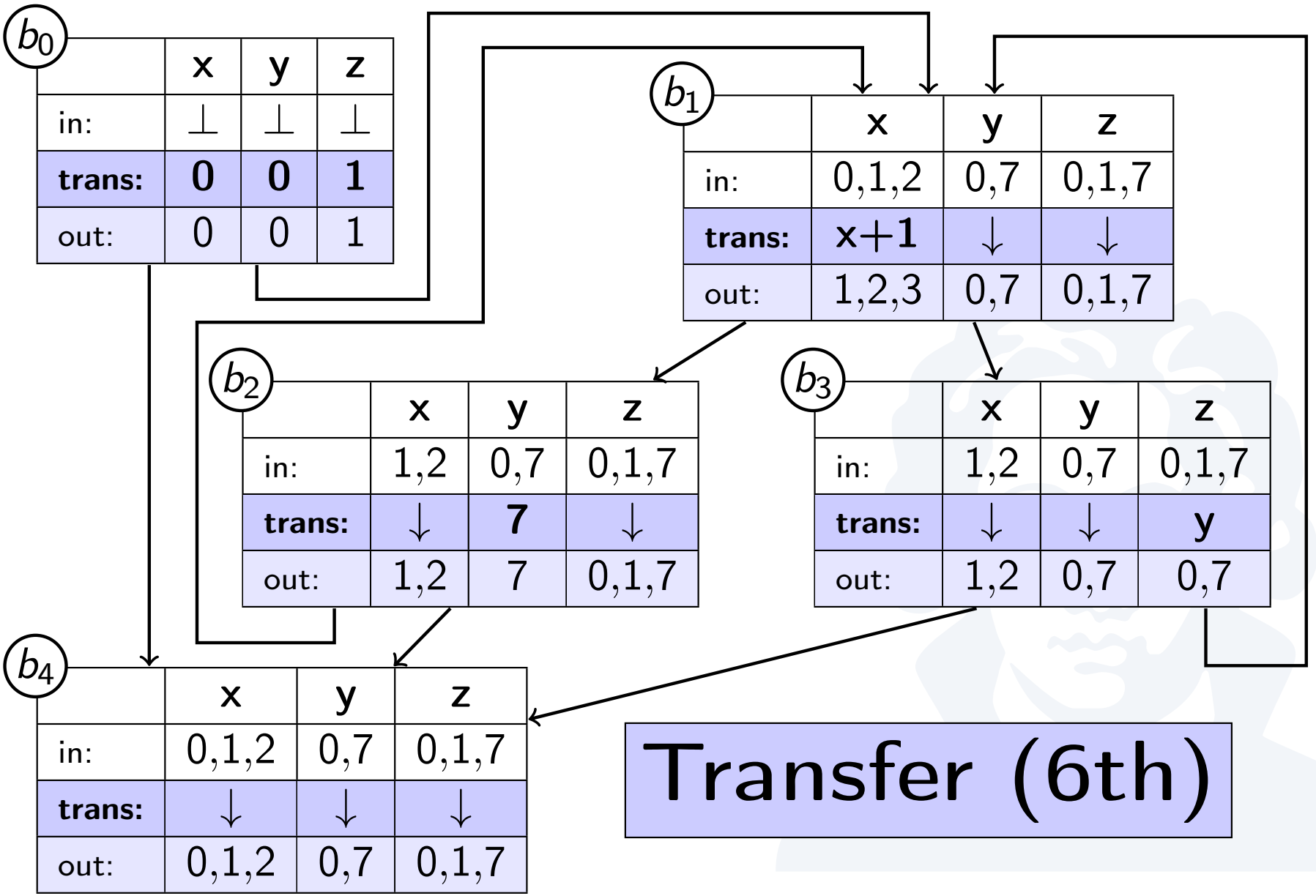
Example: Computing the Fixpoint



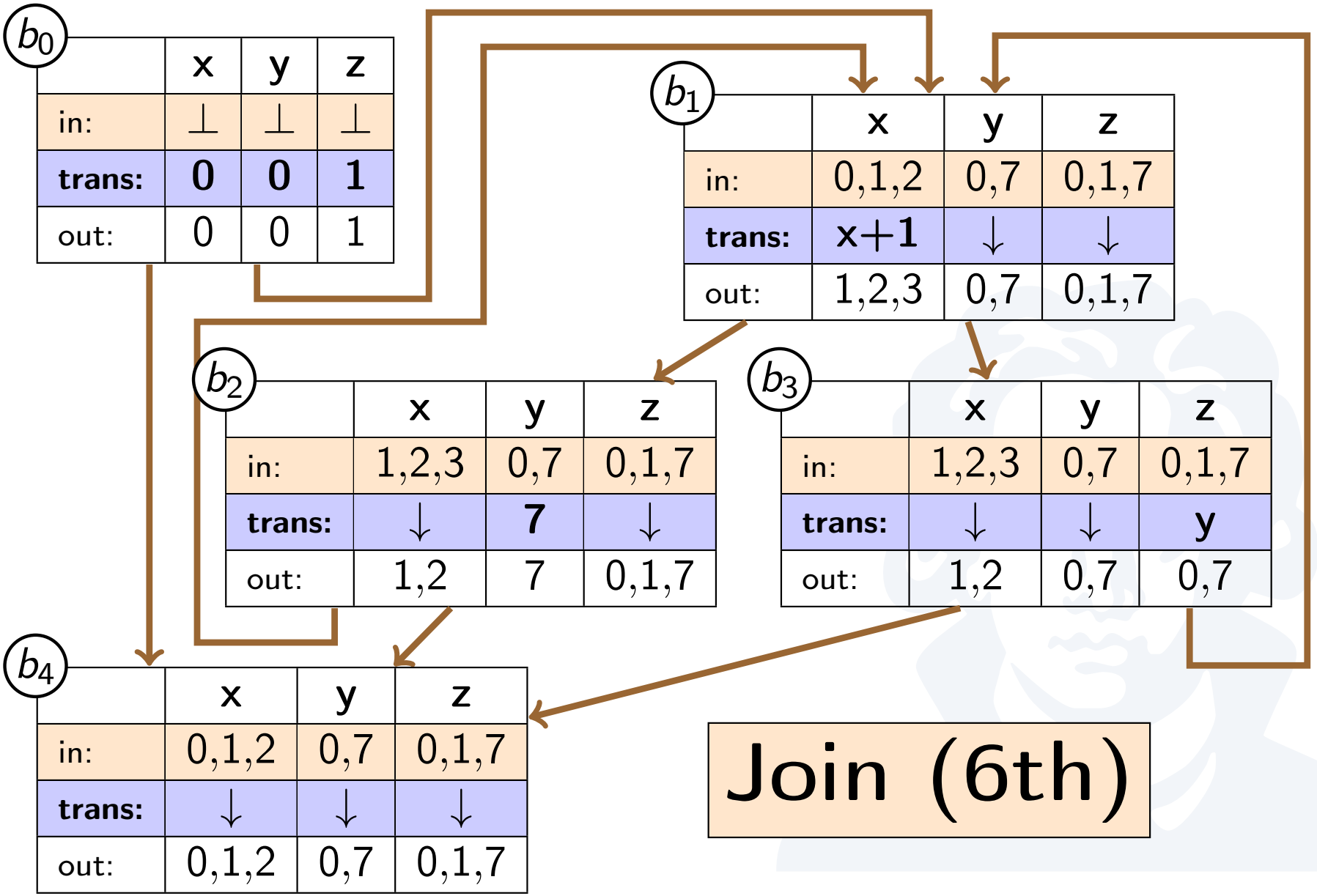
Example: Computing the Fixpoint



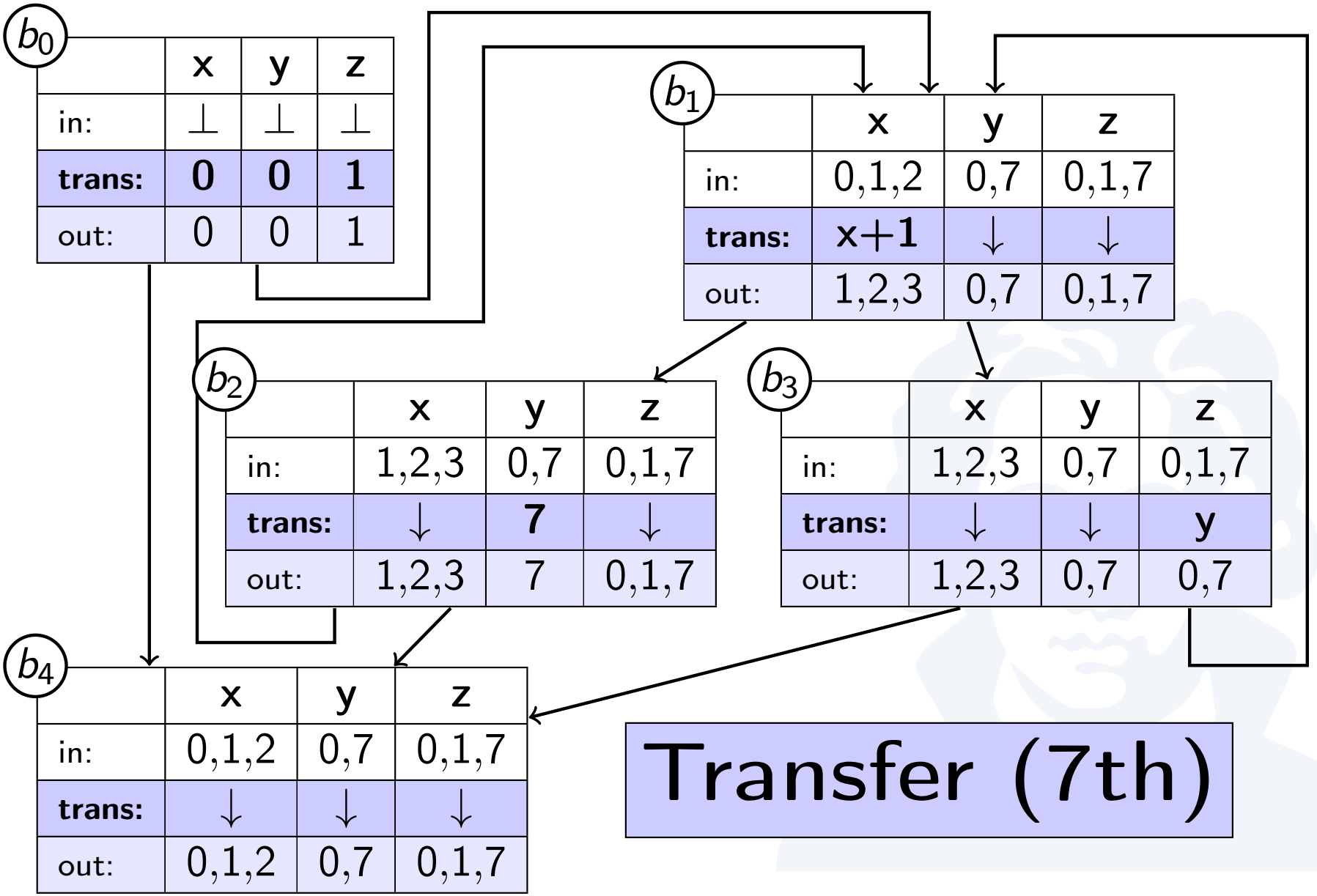
Example: Computing the Fixpoint



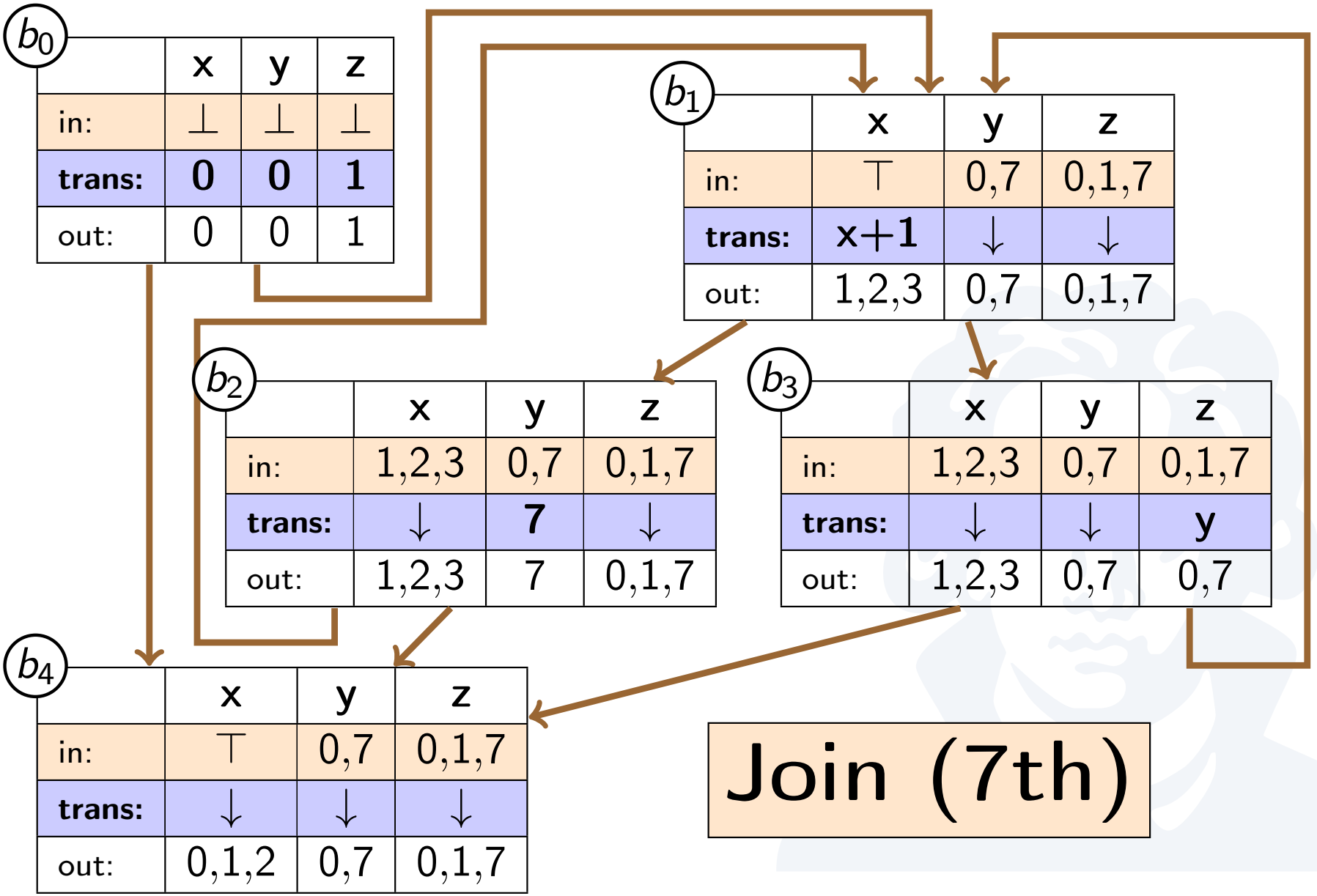
Example: Computing the Fixpoint



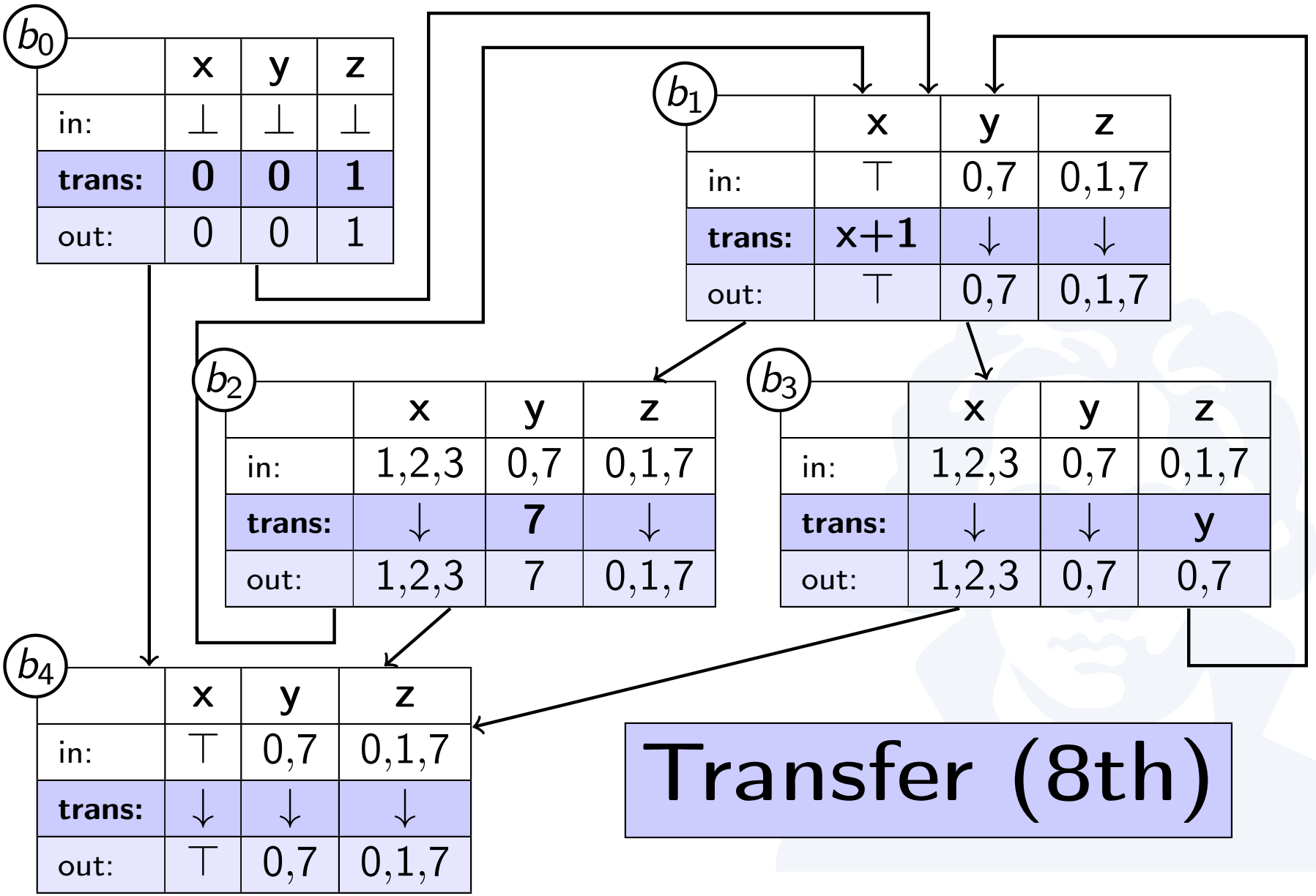
Example: Computing the Fixpoint



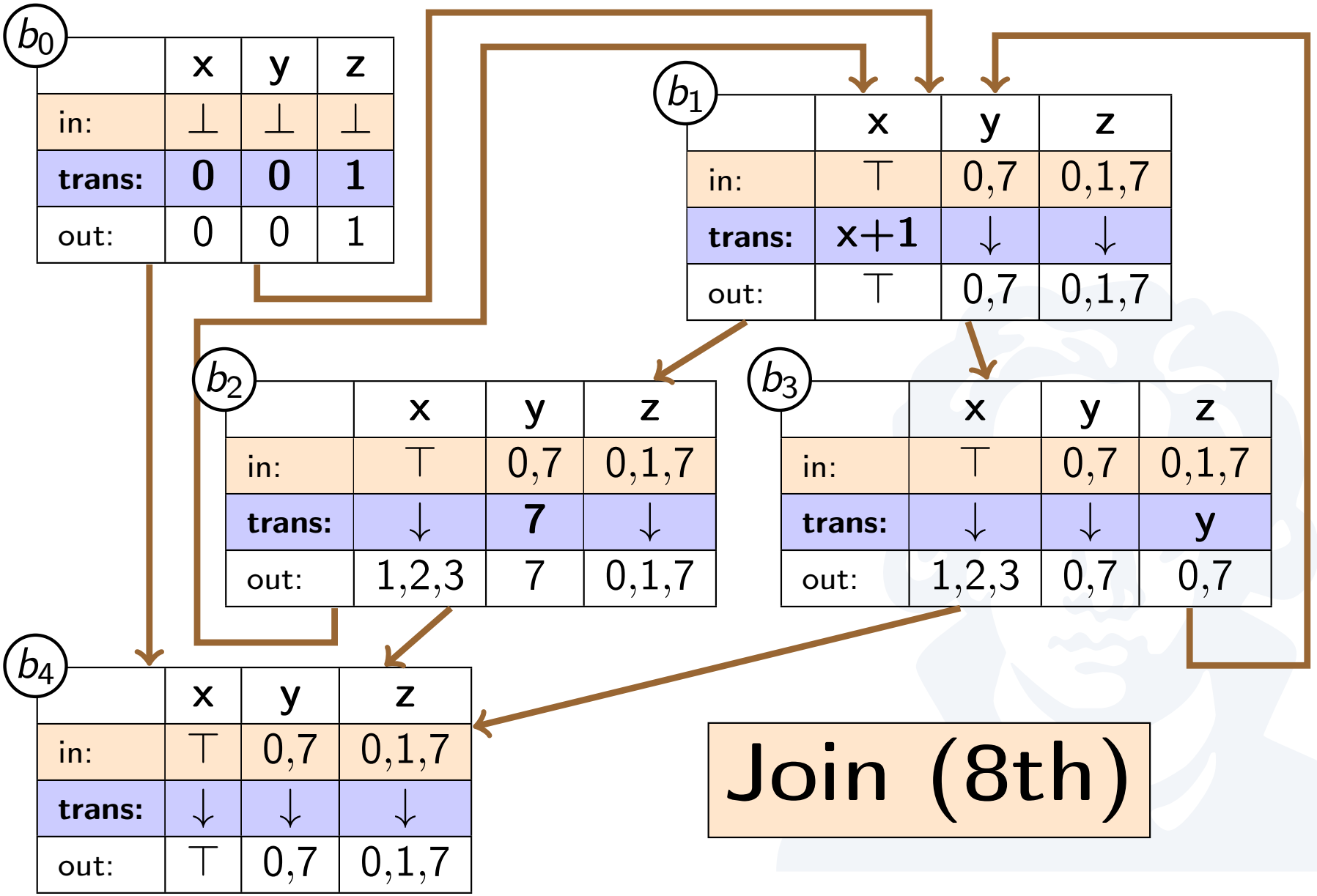
Example: Computing the Fixpoint



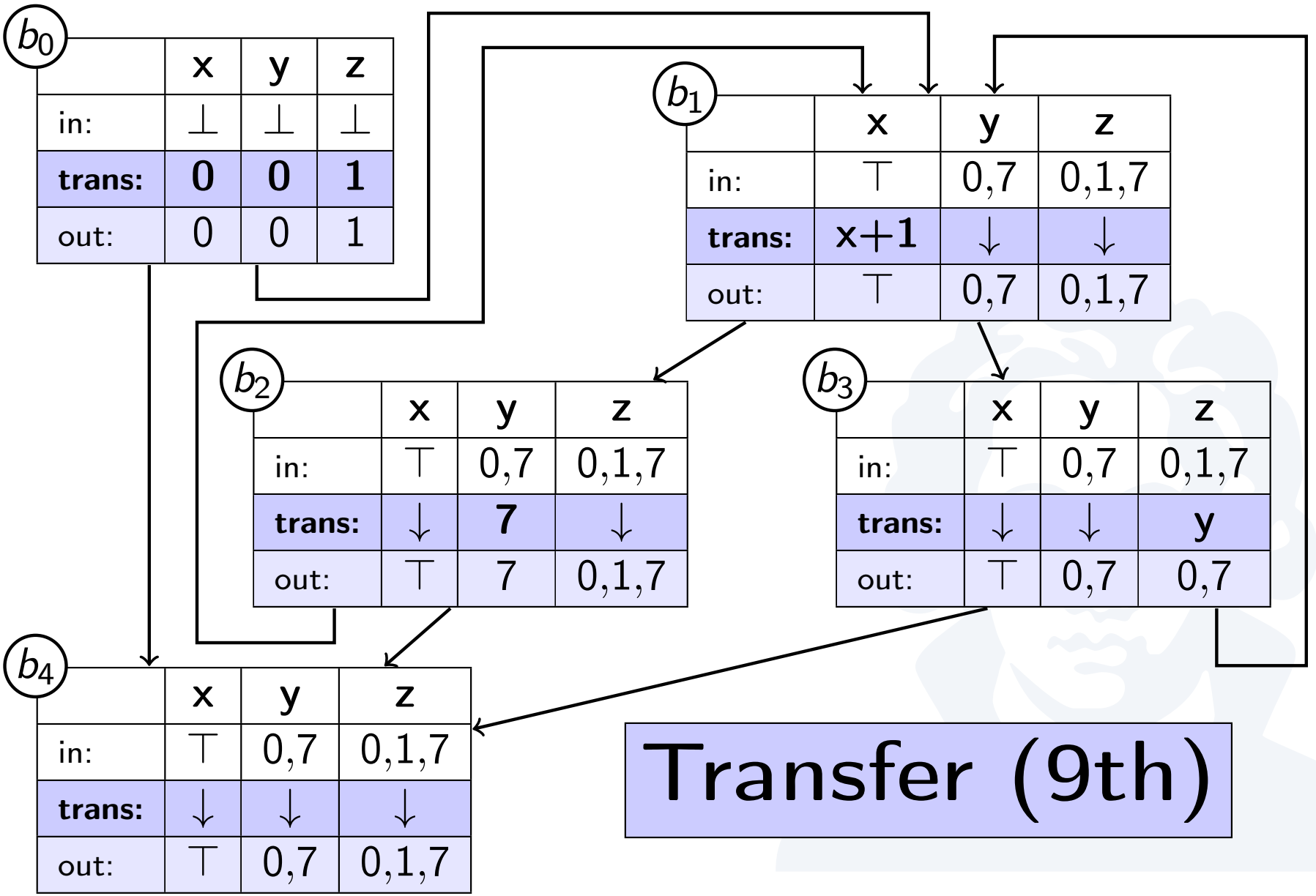
Example: Computing the Fixpoint



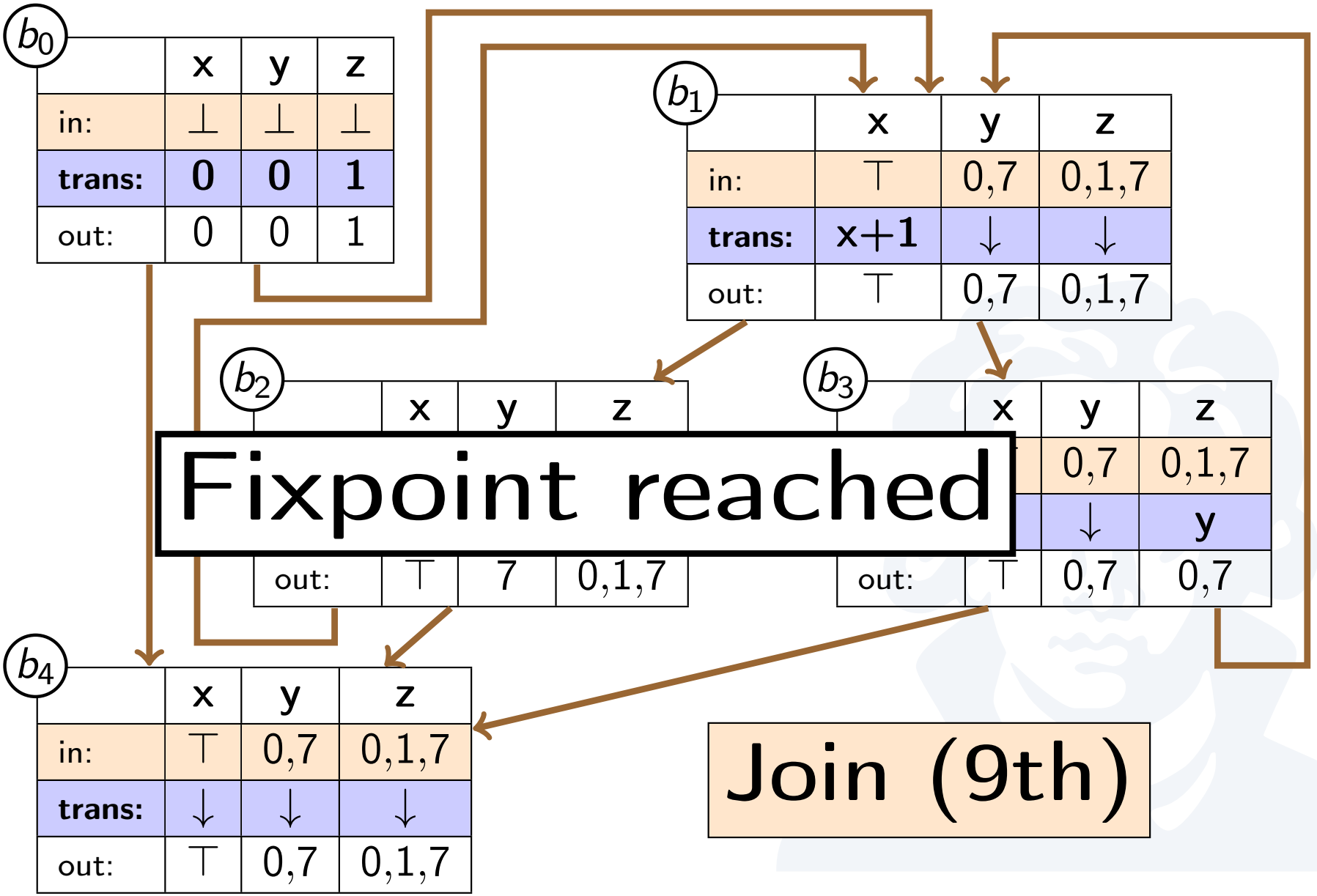
Example: Computing the Fixpoint



Example: Computing the Fixpoint



Example: Computing the Fixpoint



Example: Conclusion

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

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

return x, y, z;
```

- 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 $\top$

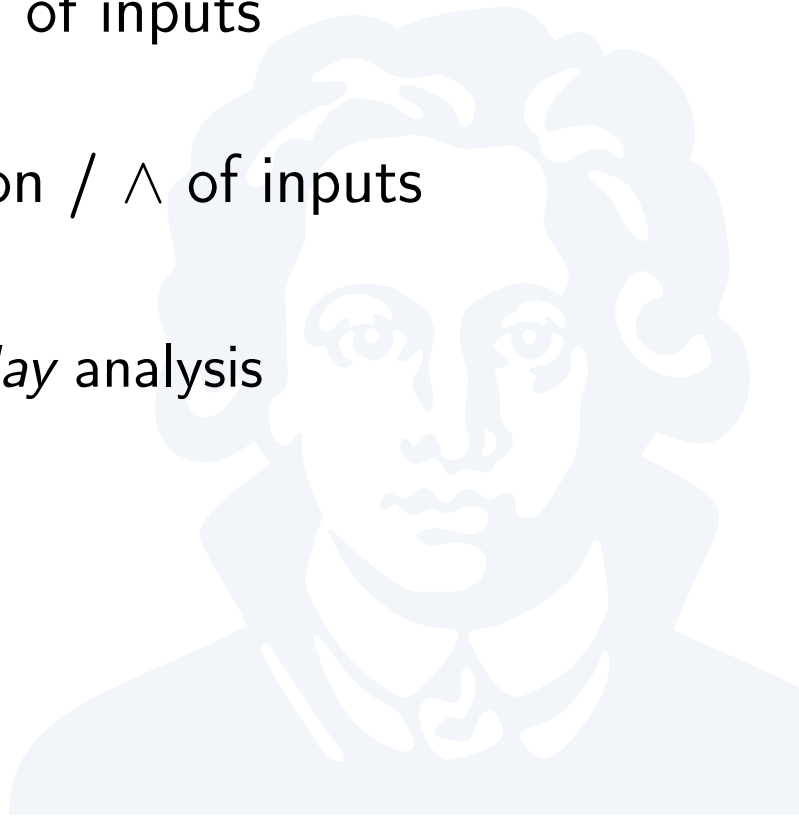
Alternatives are possible:

- Faster $x + 1$: always $\top$
- More precise conditions (flow-sensitive analysis)
- Bigger/smaller domain

"May" and "Must" analyses

- *May analysis*: Construct union / $\vee$ of inputs
  More information $\rightarrow$ closer to $\top$
- *May analysis*: Construct intersection / $\wedge$ of inputs
  More information $\rightarrow$ closer to $\bot$

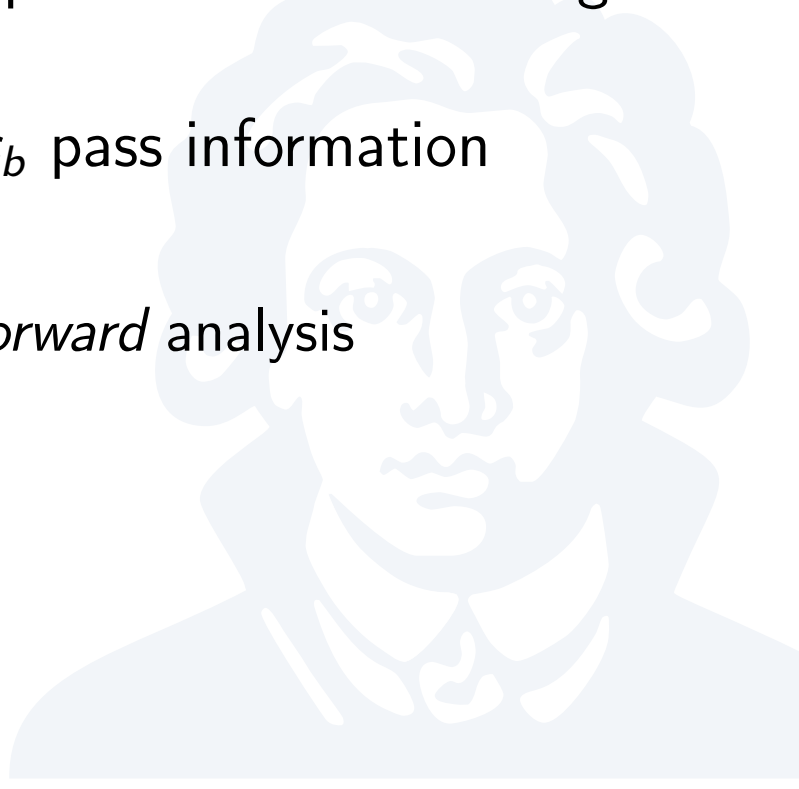
Reaching definitions is an example of a *May* analysis



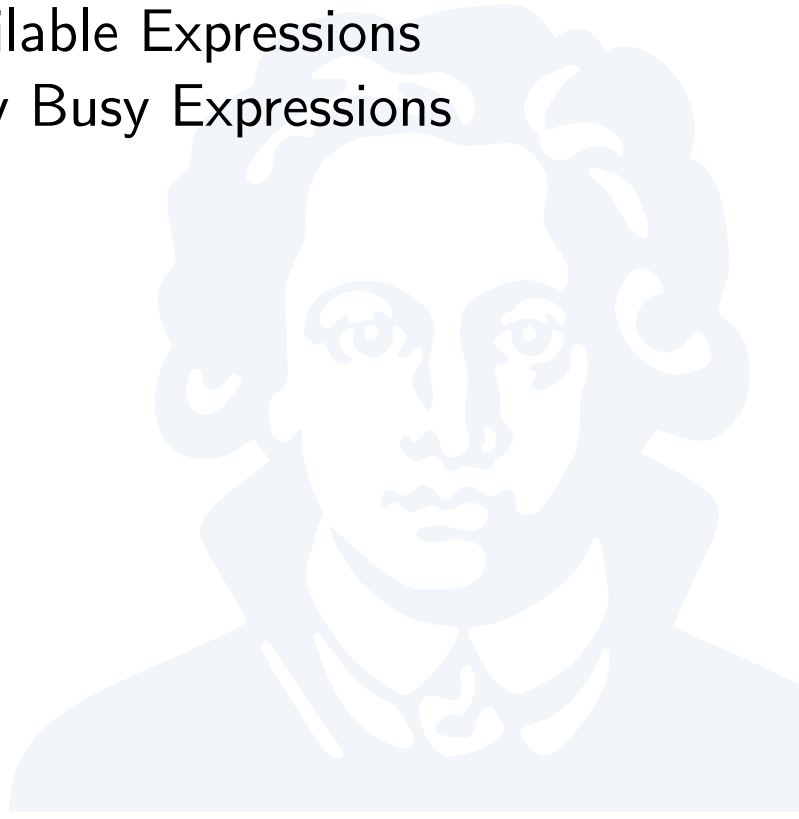
Forward and Backward analyses

- *Forward analysis*: $join_b$ and $trans_b$ pass information along control flow graph
- *Backward analysis*: $join_b$ and $trans_b$ pass information along *inverse* control flow graph

Reaching definitions is an example of a *Forward* analysis

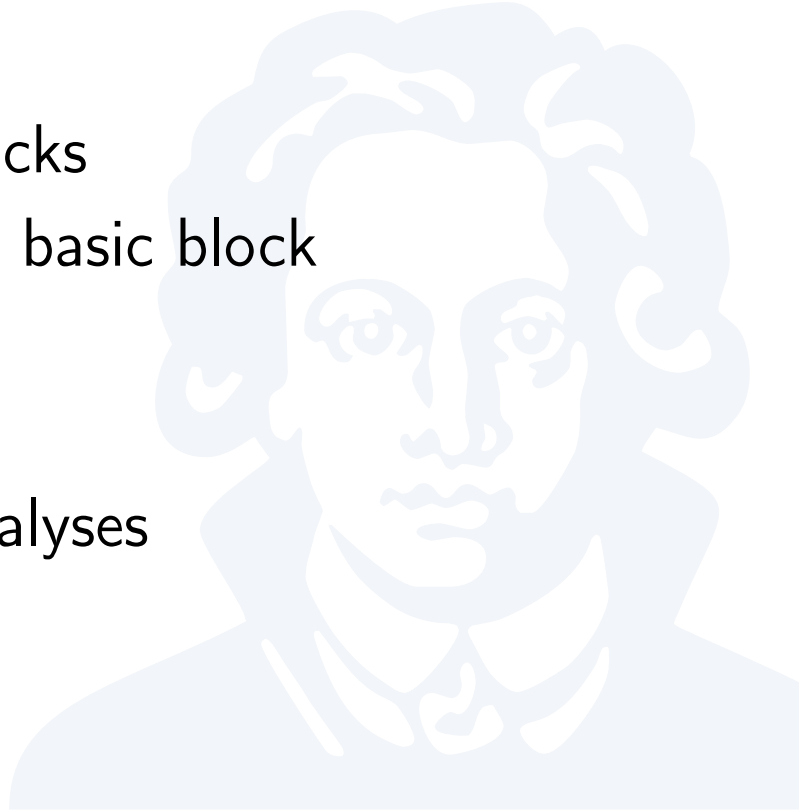


	May	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions



Summary: Data-Flow Analysis

- Analyses properties of variables
- Analysis lattice encodes the viable values we consider
- Operates on control-flow graph
- $join_b$ merges inputs from input blocks
- $trans_b$ transfers information within basic block
- Compute fixed point
- Distinction: May/Must analyses
- Distinction: Forward/Backward analyses



Constraint-Based Analysis

- Idea: separate
  - *constraint generation* and
  - *constraint solution*
- Very generic strategy
- Easy to scale up and down
- Very little general structure

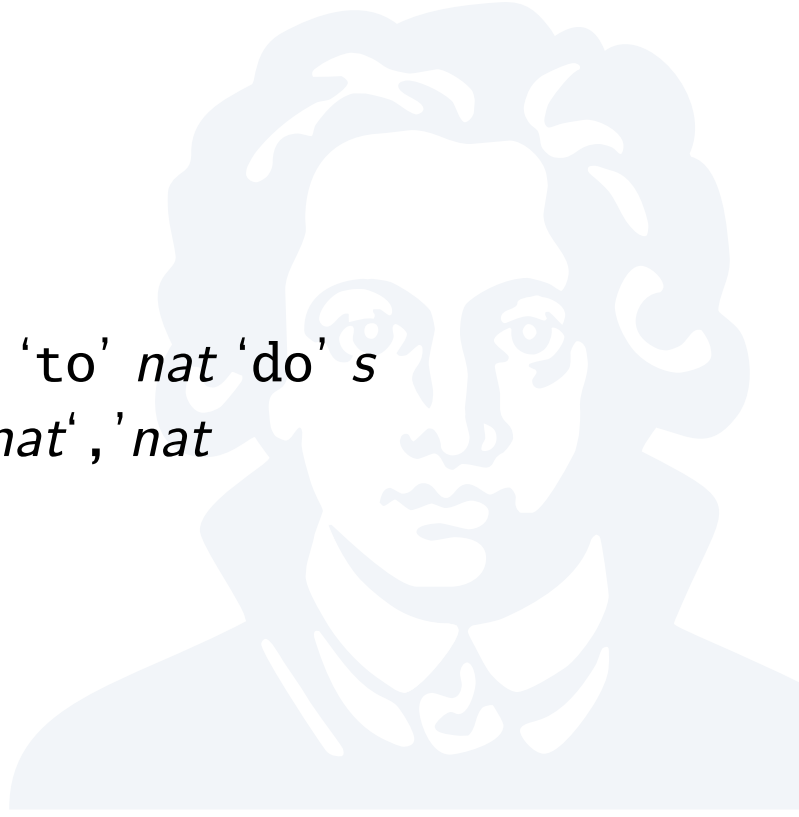
Abstract Interpretation

- Idea: use *alternative denotational semantics*, $[\![-]\!]_{\mathcal{A}}$
- $[\![-]\!]_{\mathcal{A}}$ is an *abstraction* over $[\![-]\!]$
- $[\![-]\!]_{\mathcal{A}}$ works on *abstract domain* that *simulates* real semantics

Consider the following language:

$$
\begin{array}{rcl}
s & ::= & \mathit{name}\text{`:='}e \\
 & | & \text{`begin'}\langle s\rangle\text{`;'}\langle s\rangle\text{`end'} \\
 & | & \text{`for'}\ \mathit{name}\ \text{`from'}\ \mathit{nat}\ \text{`to'}\ \mathit{nat}\ \text{`do'}\ s \\
 & | & \text{`read'}\mathit{name}\text{`between'}\mathit{nat}\text{`,'}\mathit{nat}
\end{array}
$$

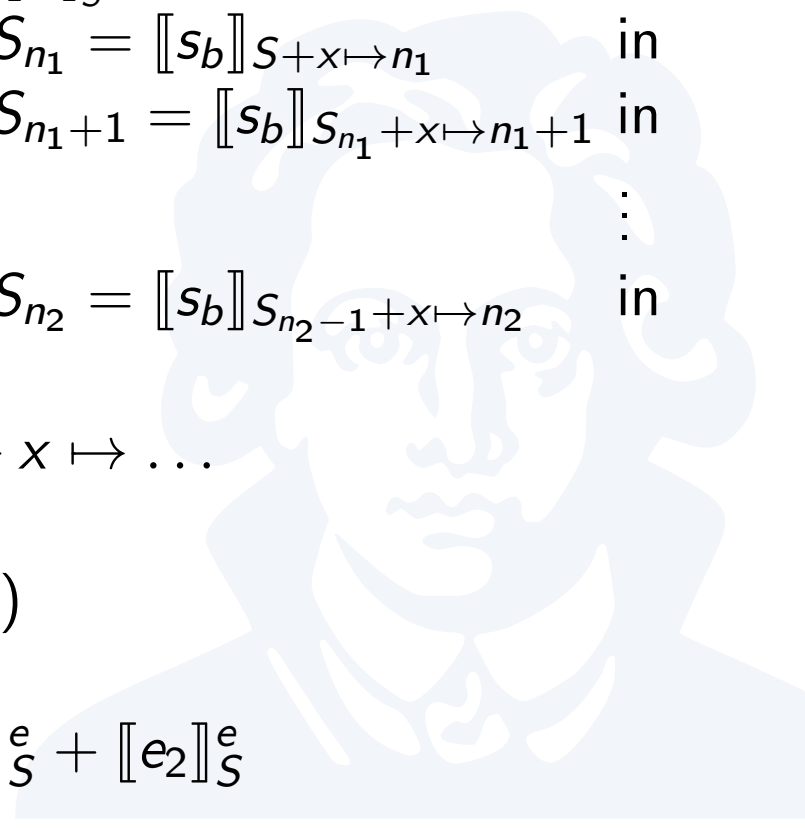
$$
\begin{array}{rcl}
e & ::= & \mathit{name} \\
 & | & \mathit{nat} \\
 & | & \langle e\rangle\text{`+'}\langle e\rangle
\end{array}
$$



Denotational Semantics

$[\![-]\!]^e$ for expressions ($\rightarrow$ values), $[\![-]\!]^s$ for statements ($\rightarrow$ states).

$$
\begin{array}{lcl}
[\![x := e]\!]^s_S & = & S + x \mapsto [\![e]\!]^e_S \\
[\![\textbf{begin } s_1; s_2 \textbf{ end}]\!]^s_S & = & [\![s_2]\!]^s_{[\![s_1]\!]^s_S} \\
[\![\textbf{for } x \textbf{ from } n_1 \textbf{ to } n_2 \textbf{ do } s_b]\!]^s_S & = & \text{let } S_{n_1} = [\![s_b]\!]_{S + x \mapsto n_1} \text{ in} \\
 & & \text{let } S_{n_1+1} = [\![s_b]\!]_{S_{n_1} + x \mapsto n_1+1} \text{ in} \\
 & & \vdots \\
 & & \text{let } S_{n_2} = [\![s_b]\!]_{S_{n_2-1} + x \mapsto n_2} \text{ in} \\
 & & S_{n_2} \\
[\![\textbf{read } x \textbf{ between } n_1, n_2]\!]^s_S & = & S + x \mapsto \ldots \\
 & & \\
[\![x]\!]^e_S & = & S(e) \\
[\![n]\!]^e_S & = & n \\
[\![e_1 + e_2]\!]^e_S & = & [\![e_1]\!]^e_S + [\![e_2]\!]^e_S
\end{array}
$$

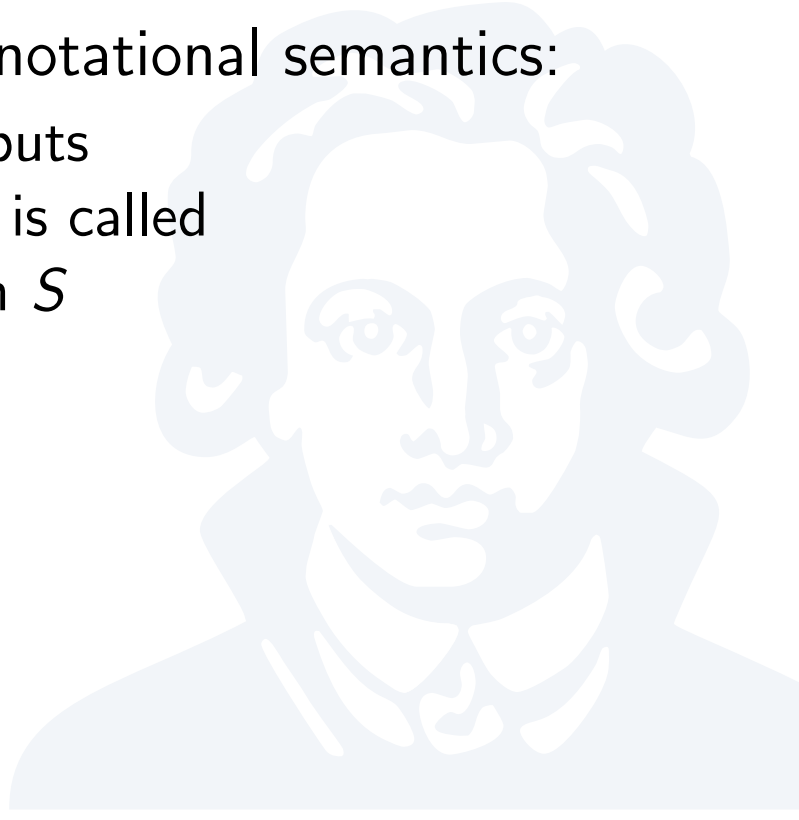


Denotational Semantics of inputs

Quick aside:

- To model inputs (as in **read** in denotational semantics:
  - Pass in (abstract) list of user inputs
  - 'Consume' input whenever **read** is called
  - Thread through program as with S

Omitted for brevity.



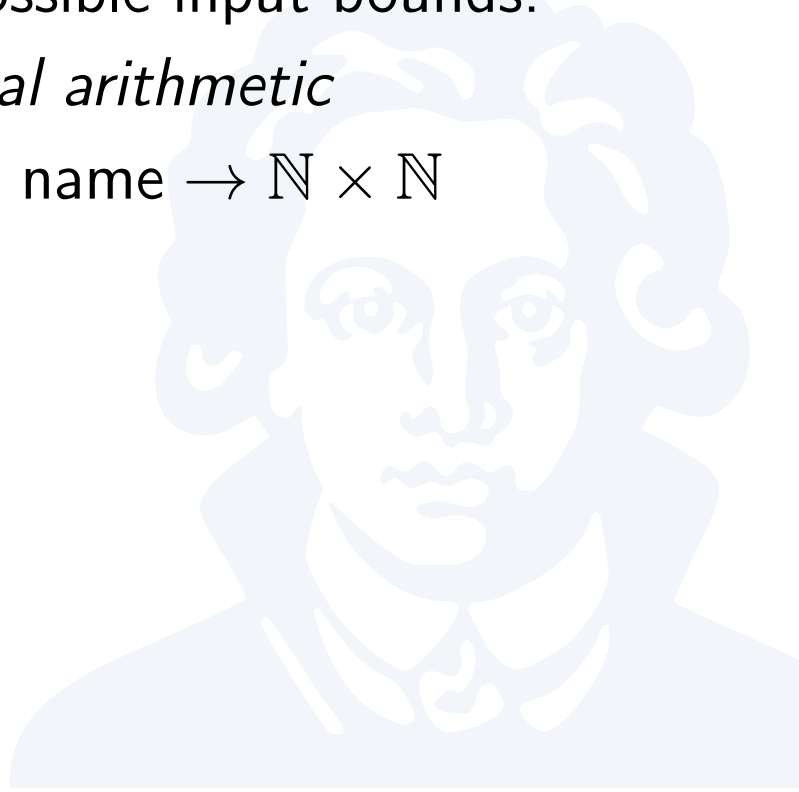
Abstract Domain Semantics: An Example

How can we make statements about such a program without having *concrete* user inputs?

- Our **read** operation tells us the possible input bounds:
- Abstract interpretation over *interval arithmetic*
- For states S for $[\![-]\!]^e$ we have: $S : \text{name} \to \mathbb{N} \times \mathbb{N}$
- Notation: $[\text{min}, \text{max}]$

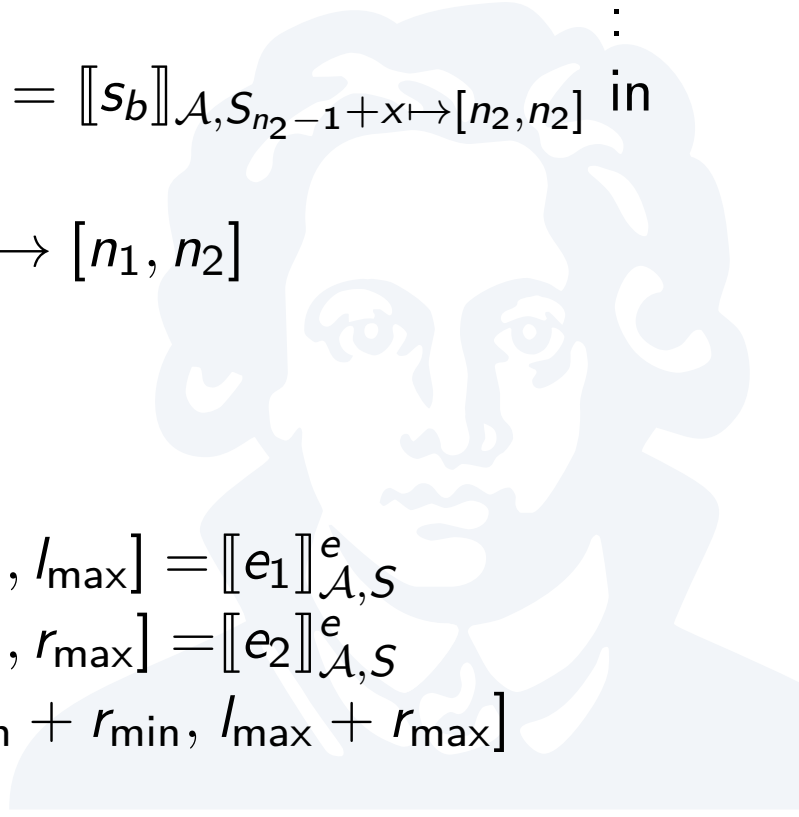
Example: x:=2 gives

$$S(x) = [2, 2]$$



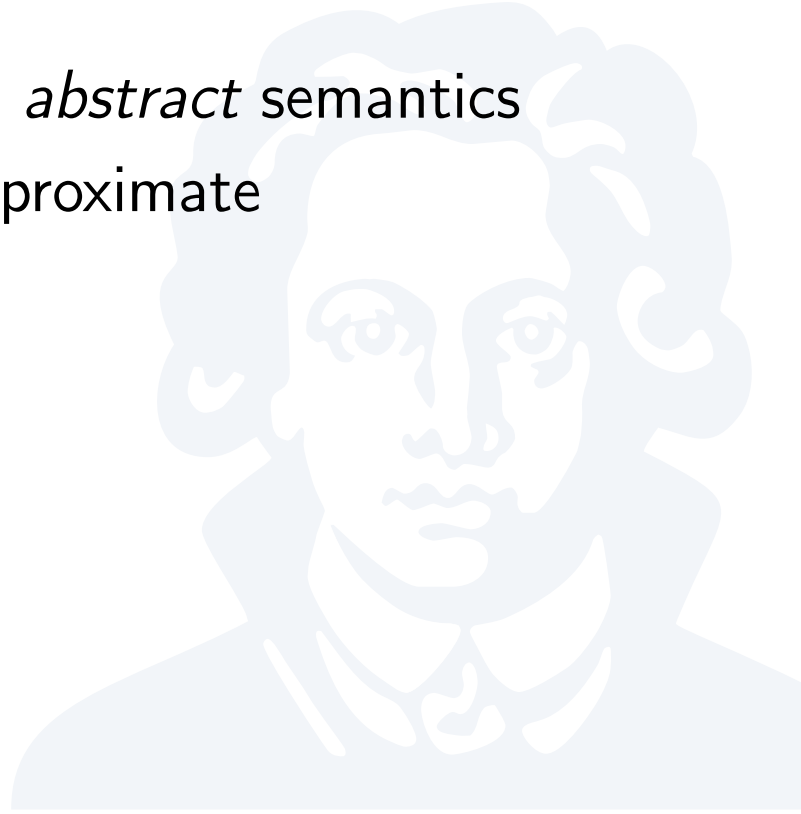
Formalising interval arithmetic

$$
\begin{array}{lcl}
[\![x := e]\!]^s_{\mathcal{A},S} & = & S + x \mapsto [\![e]\!]^e_{\mathcal{A},S} \\
[\![\textbf{begin } s_1; s_2 \textbf{ end}]\!]^s_{\mathcal{A},S} & = & [\![s_2]\!]^s_{\mathcal{A},[\![s_1]\!]^s_{\mathcal{A},S}} \\
[\![\textbf{for } x \textbf{ from } n_1 \textbf{ to } n_2 \textbf{ do } s_b]\!]^s_{\mathcal{A},S} & = & \text{let } S_{n_1} = [\![s_b]\!]_{\mathcal{A},S+x\mapsto[n_1,n_1]} \text{ in} \\
 & & \vdots \quad\quad \vdots \\
 & & \text{let } S_{n_2} = [\![s_b]\!]_{\mathcal{A},S_{n_2-1}+x\mapsto[n_2,n_2]} \text{ in} \\
 & & S_{n_2} \\
[\![\textbf{read } x \textbf{ between } n_1, n_2]\!]^s_{\mathcal{A},S} & = & S + x \mapsto [n_1, n_2] \\
 & & \\
[\![x]\!]^e_{\mathcal{A},S} & = & S(e) \\
[\![n]\!]^e_{\mathcal{A},S} & = & [n, n] \\
[\![e_1 + e_2]\!]^e_{\mathcal{A},S} & = & \text{let } [l_{\min}, l_{\max}] = [\![e_1]\!]^e_{\mathcal{A},S} \\
 & & \quad [r_{\min}, r_{\max}] = [\![e_2]\!]^e_{\mathcal{A},S} \\
 & & \text{in } [l_{\min} + r_{\min}, l_{\max} + r_{\max}]
\end{array}
$$



Summary: Abstract Interpretation

- Replace denotational semantics by *abstract* semantics
- Abstract semantics should over-approximate



Analysis Qualities: Soundness

Soundness:

- Does the analysis correctly model *all* possibilities?
- Unsound $\Rightarrow$ false negatives
- Compiler optimisations *must* be fully sound
- Unsoundness is permissible for:
  - *Static checkers*: May miss some bugs
  - *Speculative* optimisations:
    - Optimised code has *guard*, validates claims at run-time
    - Revert to unoptimised behaviour otherwise

Unsoundness example:

```
int i = 2;
if (...) { i = 3; }
```

Claim: "`i` is always 3"

Analysis Qualities: Precision

Precision:

- Does the analysis model *only real* possibilities?
- Imprecise ⇒ false positives
- Imprecision may forbid some viable optimisations

Conservative analysis: sound but not 100% precise

Imprecision example:

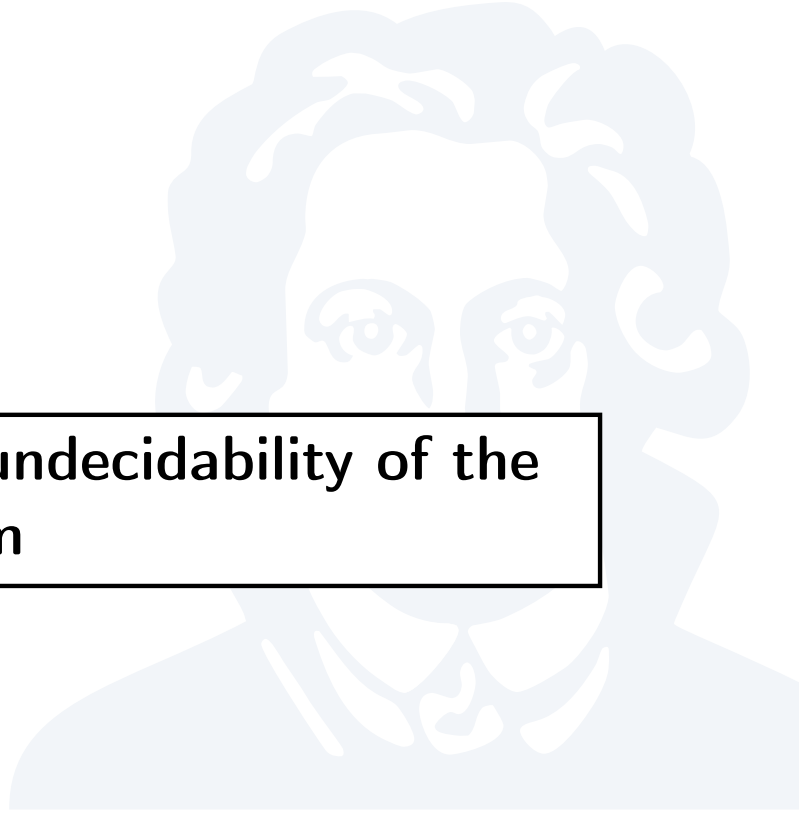
```
int i = 2;
if (...) { i = 3; }
```

Claim: "`i` is between `0` and `1000`"

Analysis Trade-offs

Analyses are trade-offs between:

- Soundness
- Precision
- Execution time



Soundness/precision bounded by undecidability of the Halting Problem

Analysis Qualities

- Intraprocedural Analyses
  Only within individual procedures
- Interprocedural Analyses
  Across procedure boundaries

Properties of particular analyses:

- Context Sensitivity
- Flow Sensitivity
- Path Sensitivity

Context Sensitivity

Consider *Calling context:*

```
int f(int i)
{
  return i;
}

...
x = 2;
y = f(x);
```

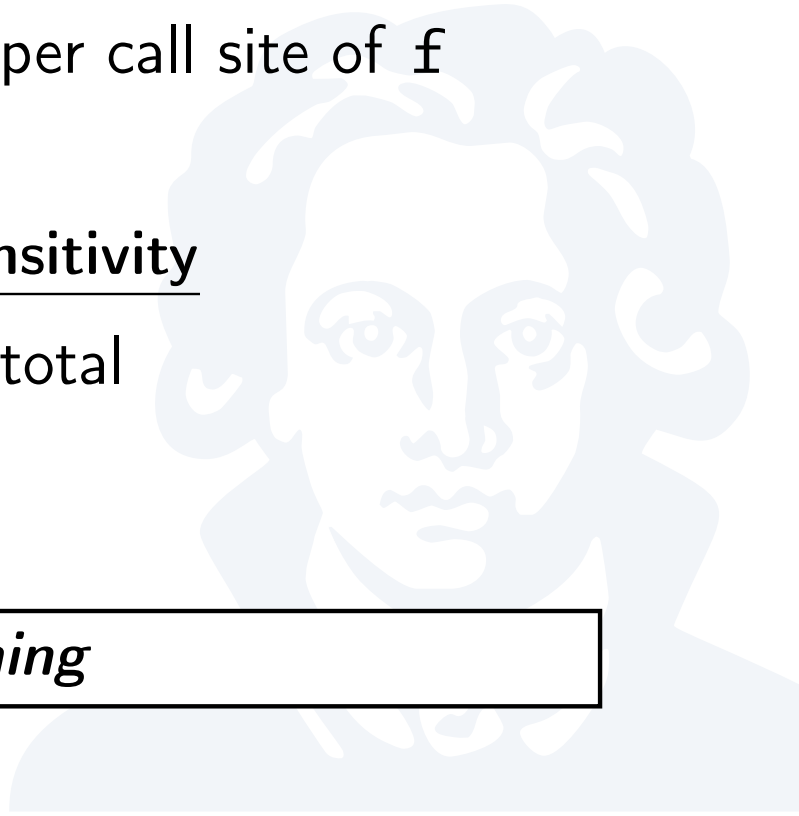
With context sensitivity

- We have one `i` per call site of `f`
- `y` is 2

Without context sensitivity

- We have one `i` total
- `y` is some `int`

Analogous to *inlining*



Flow Sensitivity

Consider *Control flow* and *Order of execution*:

```
x = 2;
y = x;
x = 3;
```

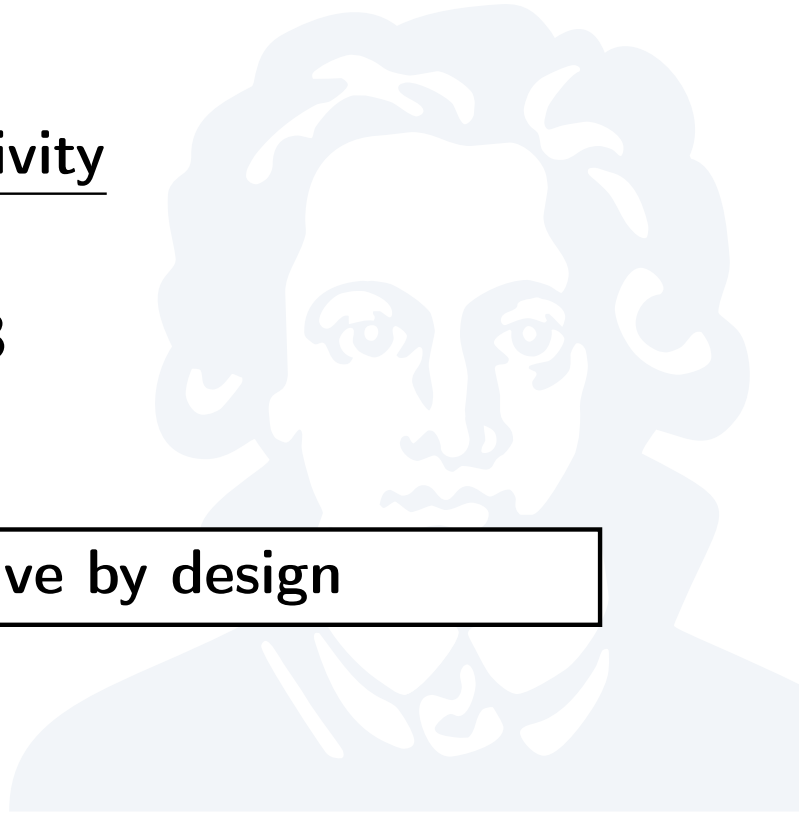
With flow sensitivity

- y is 2

Without flow sensitivity

(assuming no SSA)

- y may be 2 or 3



Flow analysis is flow sensitive by design

Path Sensitivity

Consider properties inferred from *Execution path:*

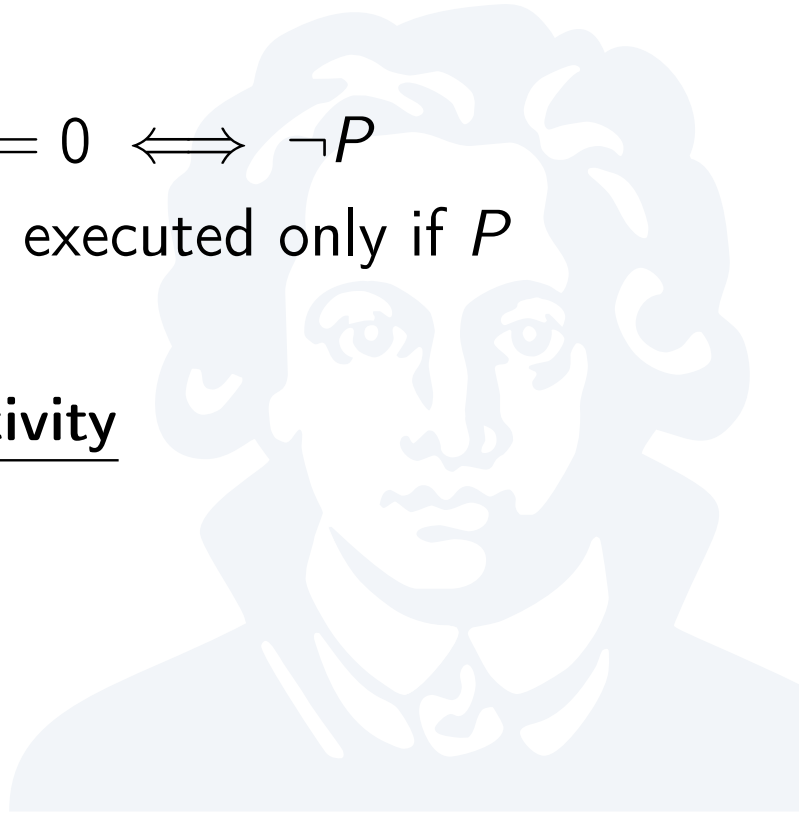
```
x = 0;
if (P) {
  x = 1;
}
y = 2;
if (P) {
  y = x; /*A*/
}
```

With path sensitivity

- $\mathtt{y} \in \{1, 2\}$
- Records that $\mathtt{x} = 0 \iff \neg P$
- Knows that `A` is executed only if P $\Rightarrow \mathtt{x} \neq 0$ at `A`

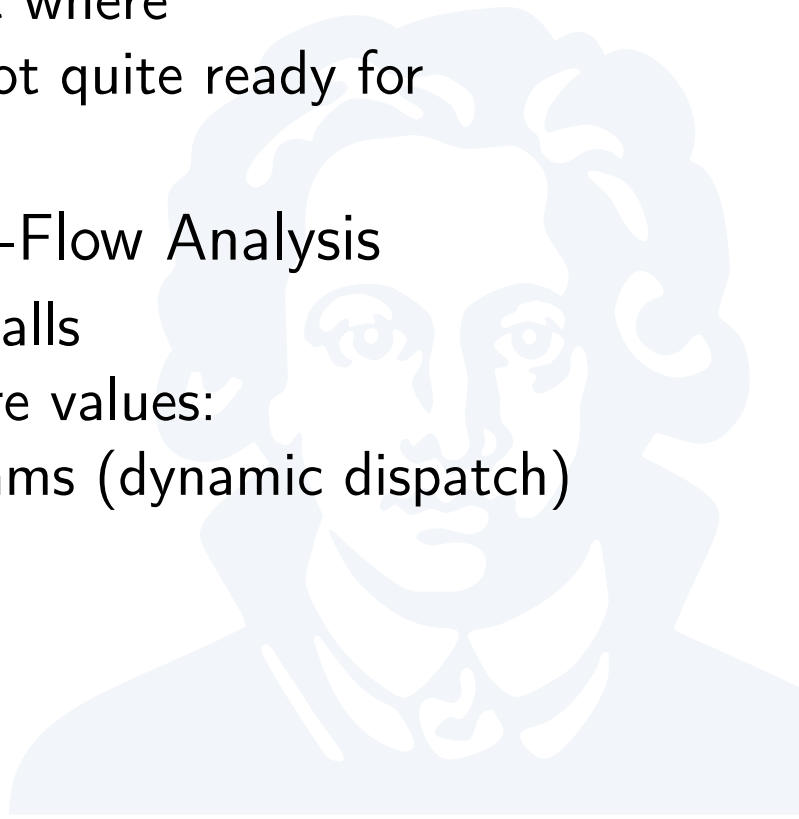
Without path sensitivity

- $\mathtt{y} \in \{0, 1, 2\}$
- Less precise



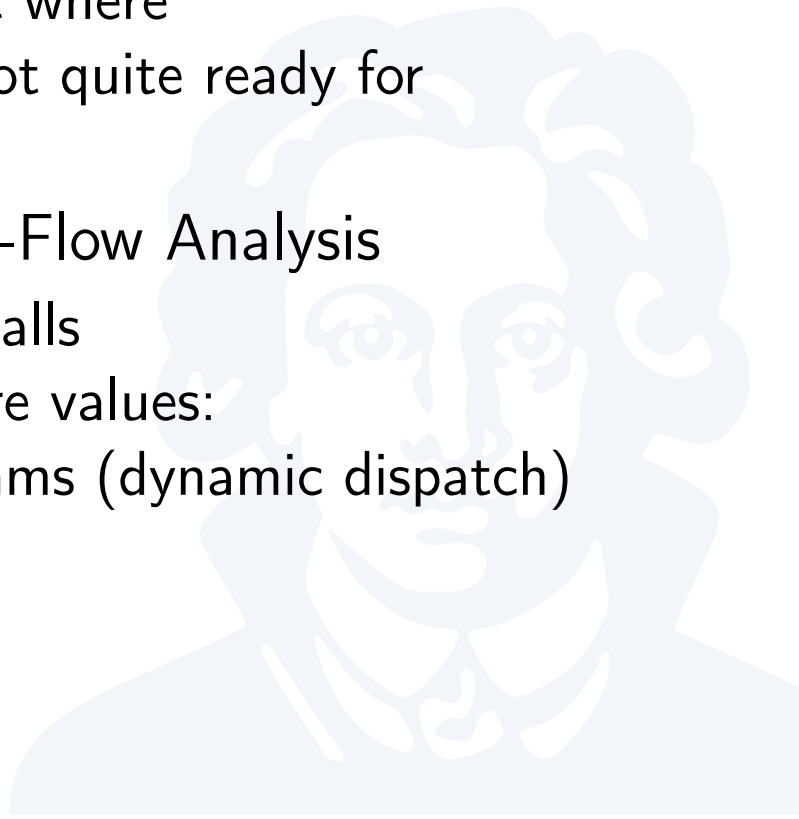
Other important analyses

- Points-to analysis
  - Determines which pointers point where
  - Substantial research area, still not quite ready for practical use
- k-CFA: (context sensitive) Control-Flow Analysis
  - Determines targets of function calls
  - Important whenever functions are values:
    Functional programs, OO programs (dynamic dispatch)



Other important analyses

- Points-to analysis
  - Determines which pointers point where
  - Substantial research area, still not quite ready for practical use
- k-CFA: (context sensitive) Control-Flow Analysis
  - Determines targets of function calls
  - Important whenever functions are values:
    Functional programs, OO programs (dynamic dispatch)



Summary

- Analysis qualities
- Precision & Soundness
- Flow-, Path-, Context-sensitivity

Kinds of analysis:

- Type and Effect analysis
- Flow Analysis
- Constraint-Based Analysis
- Abstract Interpretation

Literature

- Flemming Nielson, Hanne R. Nielson, Chris Hankin: "Principles of Program Analysis" (book)
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: "Compilers: Principles, Techniques, and Tools" (book)
- Alex Aiken: "Introduction to Set Constraint-Based Program Analysis"
- Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren: "The Program Dependence Graph and Its Use in Optimization" (ACM TOPLAS, 1987)
- Sumit Gulwani, Ashish Tiwari: "Computing Procedure Summaries for Interprocedural Analysis " (ESOP 2007)
- Ondej Lhoták , Yannis Smaragdakis, and Manu Sridharan: "Pointer Analysis", (Report from Dagstuhl Seminar 13162)

Next week:

Dynamic Program Analysis