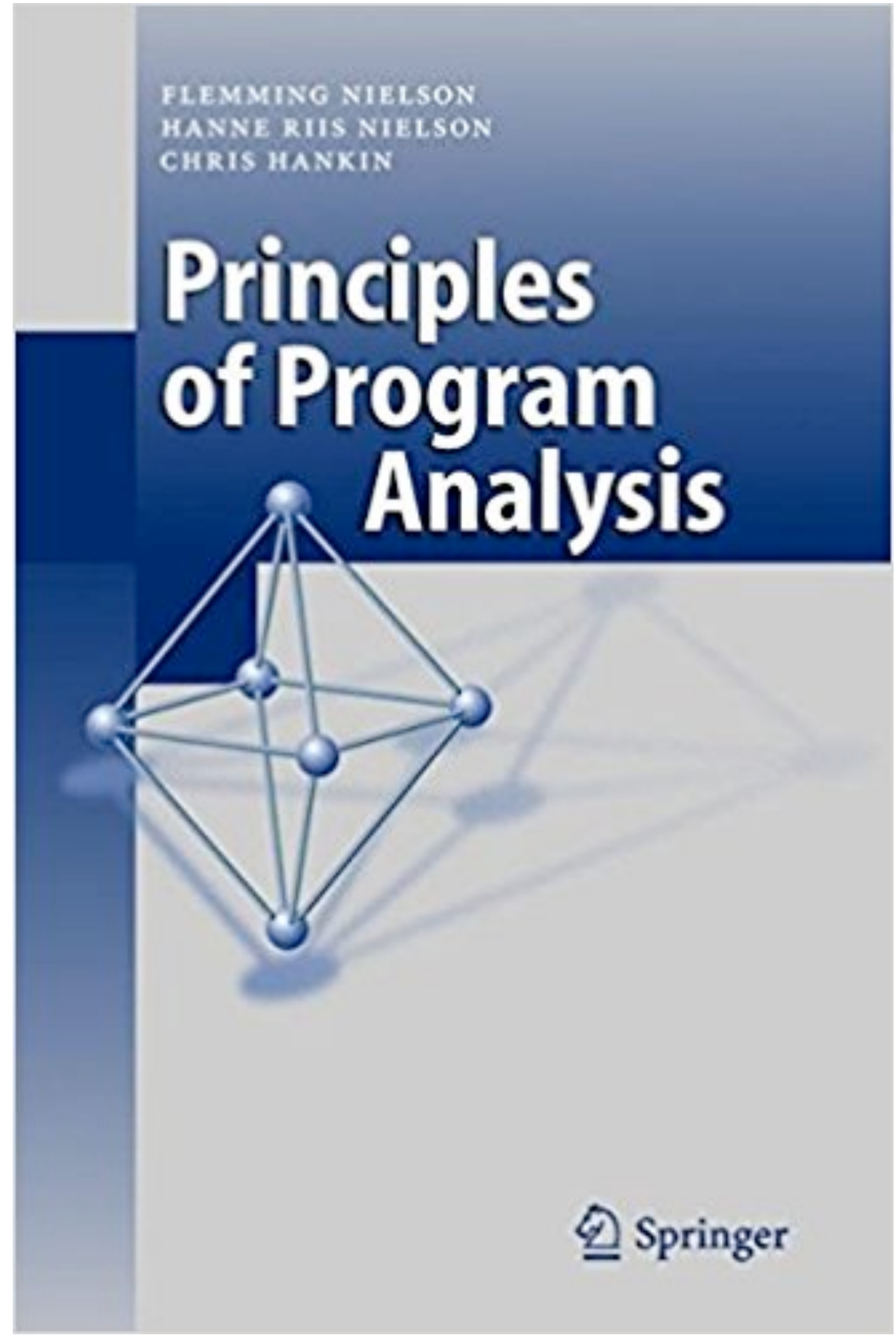
# Flow Analysis

Data-flow analysis, Control-flow analysis, Abstract interpretation, AAM

**Helpful Reading:**

Sections 1.1-1.5, 2.1

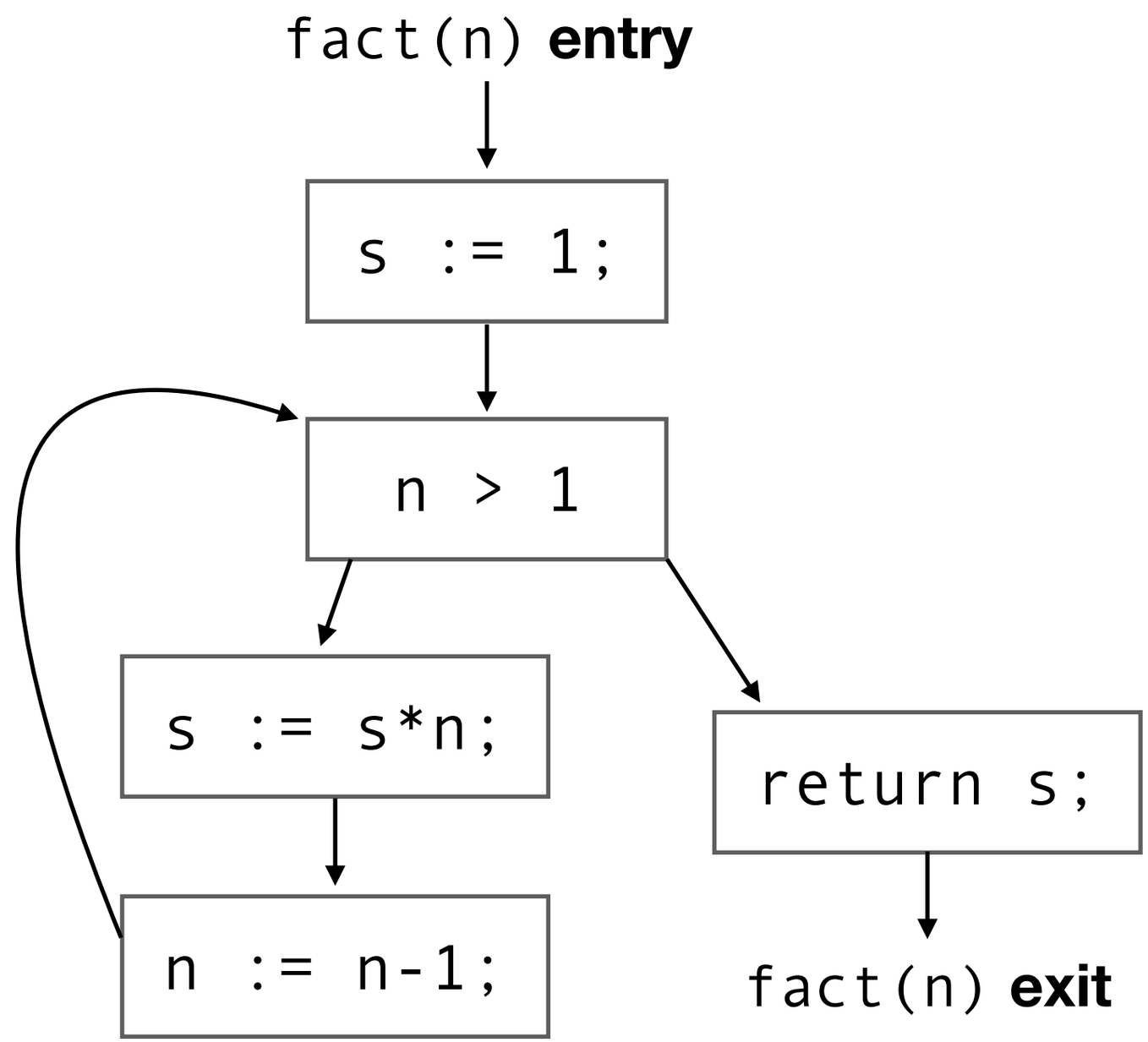


# **Data-flow analysis** (DFA)

- A framework for statically proving facts about program data.
    - Focuses on simple, finite facts about programs.
- Necessarily over- or under-approximate (*may* or *must*).
    - Conservatively considers all possible behaviors.
- Requires control-flow information; i.e., a control-flow graph.
    - If imprecise, DFA may consider infeasible code paths!
- Examples: reaching defs, available expressions, liveness,…

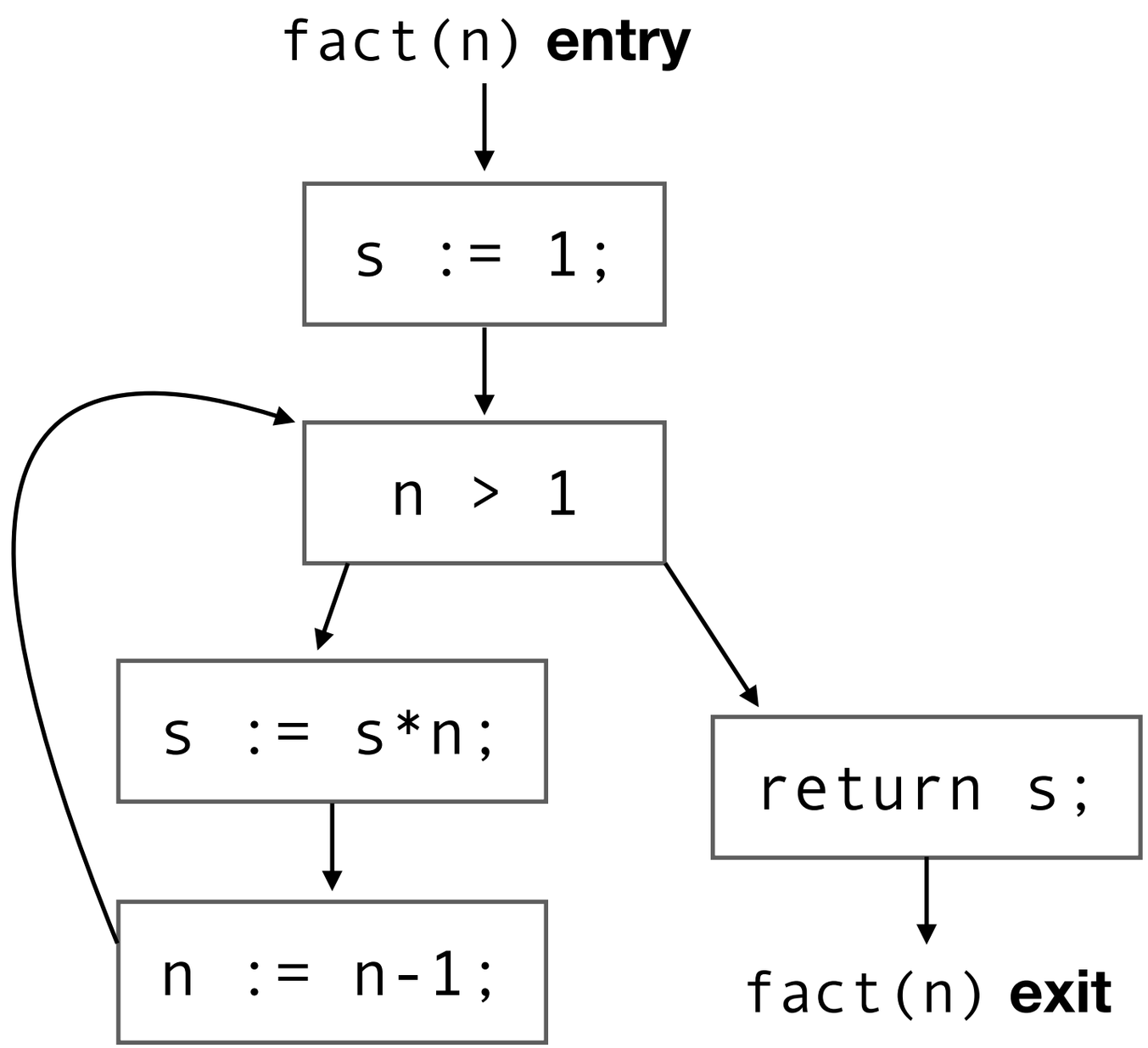
# Control-flow graphs (CFGs)

```
define fact(n : int)
{
    s := 1;
    while (n > 1)
    {
        s := s*n;
        n := n-1;
    }

    return s;
}
```



# Control-flow graphs (CFGs)

- Intraprocedural CFGs may have a single entry/exit.
- Nodes can be a full **basic block** or a single statement.
- Each block or statement has $1^+$ predecessor and $1^+$ successor (stmt or block).
- A ***fork point*** is where paths diverge and a ***join point*** is where paths come together.



# Data-flow analysis

- Computed by propagating facts ***forward*** or ***backward***.
- Computes ***may*** or ***must*** information.
- Reaching definitions/assignments (def-use info): which assignments may reach each variable reference (use).
- Liveness: which variables are still needed at each point.
- Available expressions: which expressions are already stored.
- Very busy expressions: which expressions are computed down all possible paths forward.

# Reaching defs, worklist algorithm

- Reaching definitions is a ***forward may*** analysis.
- gen(s) yields facts generated by a statement s.
- kill(s) yields facts invalidated by a statement s.
- pred(s) and succ(s) yield sets of statements preceding/succ.
- entry(s) = $\bigcup_{s' \in pred(s)} exit(s')$;  exit(s) = (entry(s) \ kill(s)) ∪ gen(s)
- gen, kill, pred, succ, are fixed for each s; exit is monotonic in entry (if entry(s) grows, exit(s) grows); entry for exits of preds.
- We iterate rules for entry/exit until reaching a ***fixed point***.

# Reaching defs, worklist algorithm

- Main idea: worklist-based fixed-point algorithm.
- All entry(s) and exit(s) are initialized to be empty.
- Add all statements to the worklist; all must be considered.
- Until the worklist is empty, remove an s from worklist:
    - Compute entry(s) as union of all exit(s’), of predecessor s’
    - Compute exit(s) from gen(s), kill(s), and entry(s)
    - If exit(s) was increased, add all succ(s) to the worklist
- (This version is for *forward may* kill/gen analyses.)

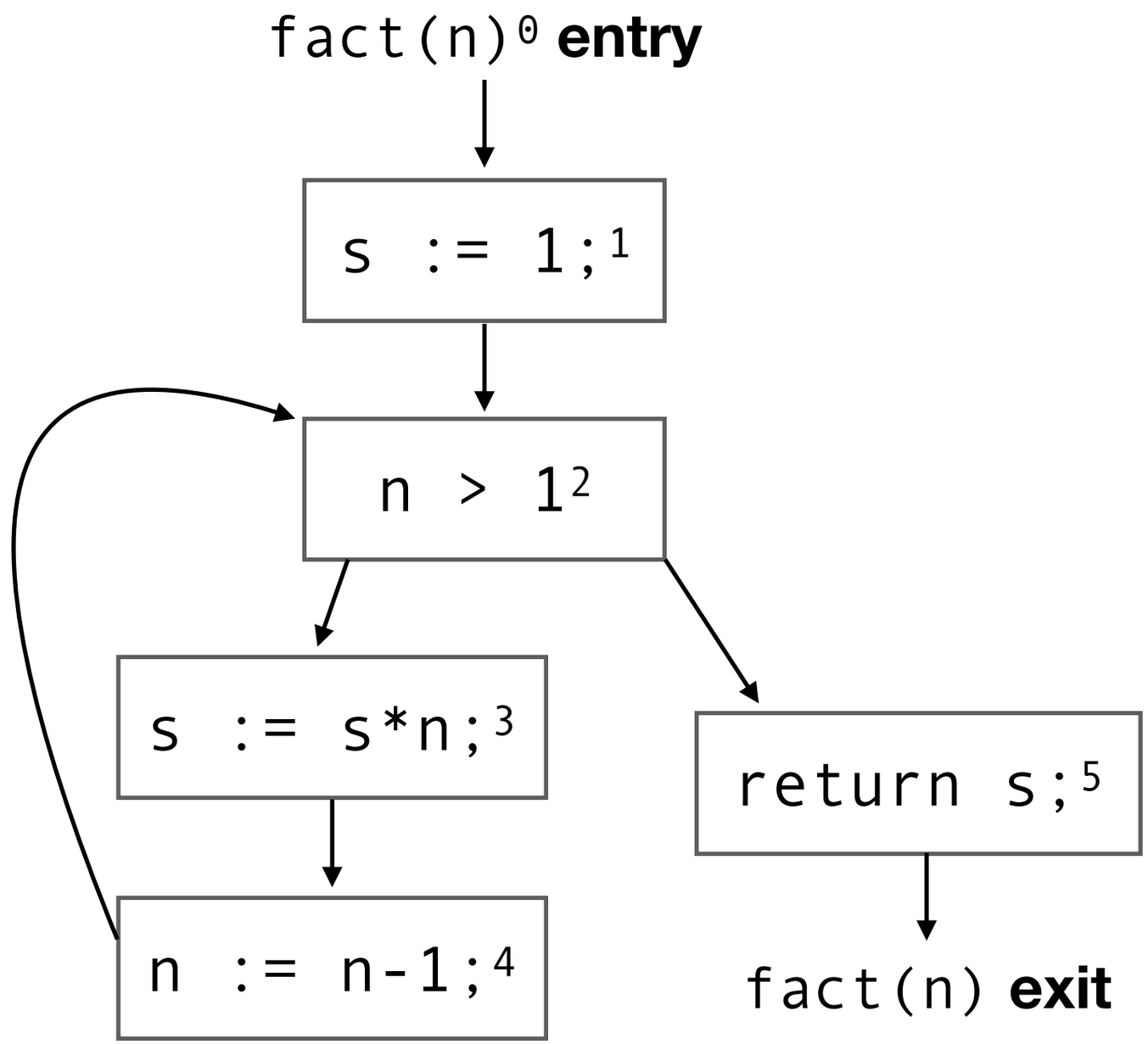
# *Reaching defs*, worklist algorithm

```
exit(s) = ∅
W = all statements s  //worklist
while W not empty:
   remove s from W

   entry(s) = ∪_{s'∈pred(s)} exit(s')

   update = (entry(s) \ kill(s)) ∪ gen(s)
   if update != exit(s):
      exit(s) = update

      W = W ∪ succ(s)
```

*Forward may* analysis

# *Reaching definitions* analysis

| Stmt | GEN | KILL |
|---|---|---|
| `s := 1;`$^{1}$ | `(s,1)` | `(s,*)` |
| `n > 1`$^{2}$ | | |
| `s := s*n;`$^{3}$ | `(s,3)` | `(s,1)`<br>`(s,3)` |
| `n := n-1;`$^{4}$ | `(n,4)` | `(n,0)`<br>`(n,4)` |
| `return s;`$^{5}$ | | |



# *Reaching definitions* analysis

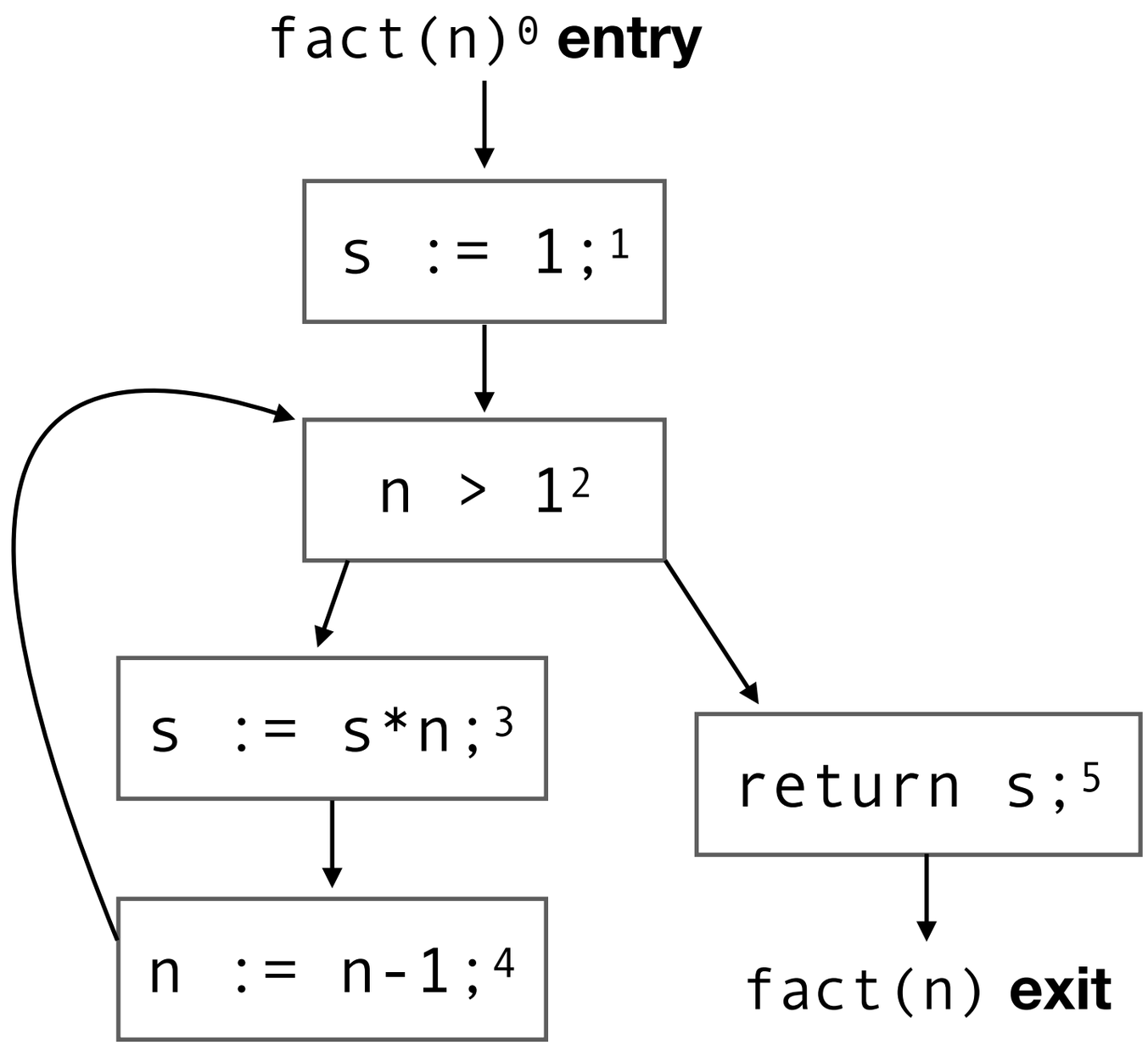
(n,0)

(n,0) (s,1) (n,4) (s,3)

(n,0) (s,1) (n,4) (s,3)

(n,0) (s,3) (n,4)

(n,4) (s,3)

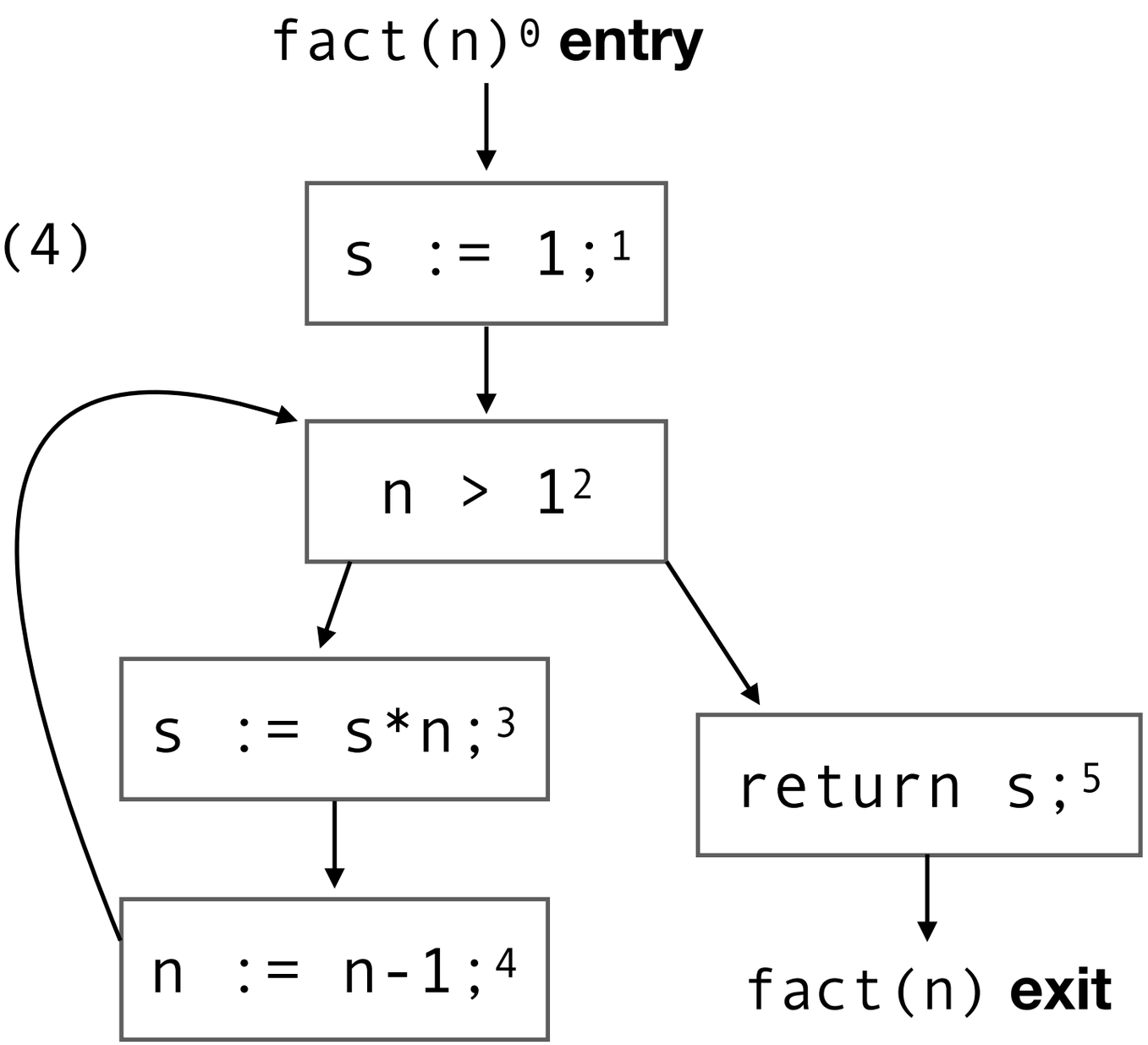


# *Reaching definitions* analysis

$all(x) = \{(x,\ell) \mid \forall \ell\}$

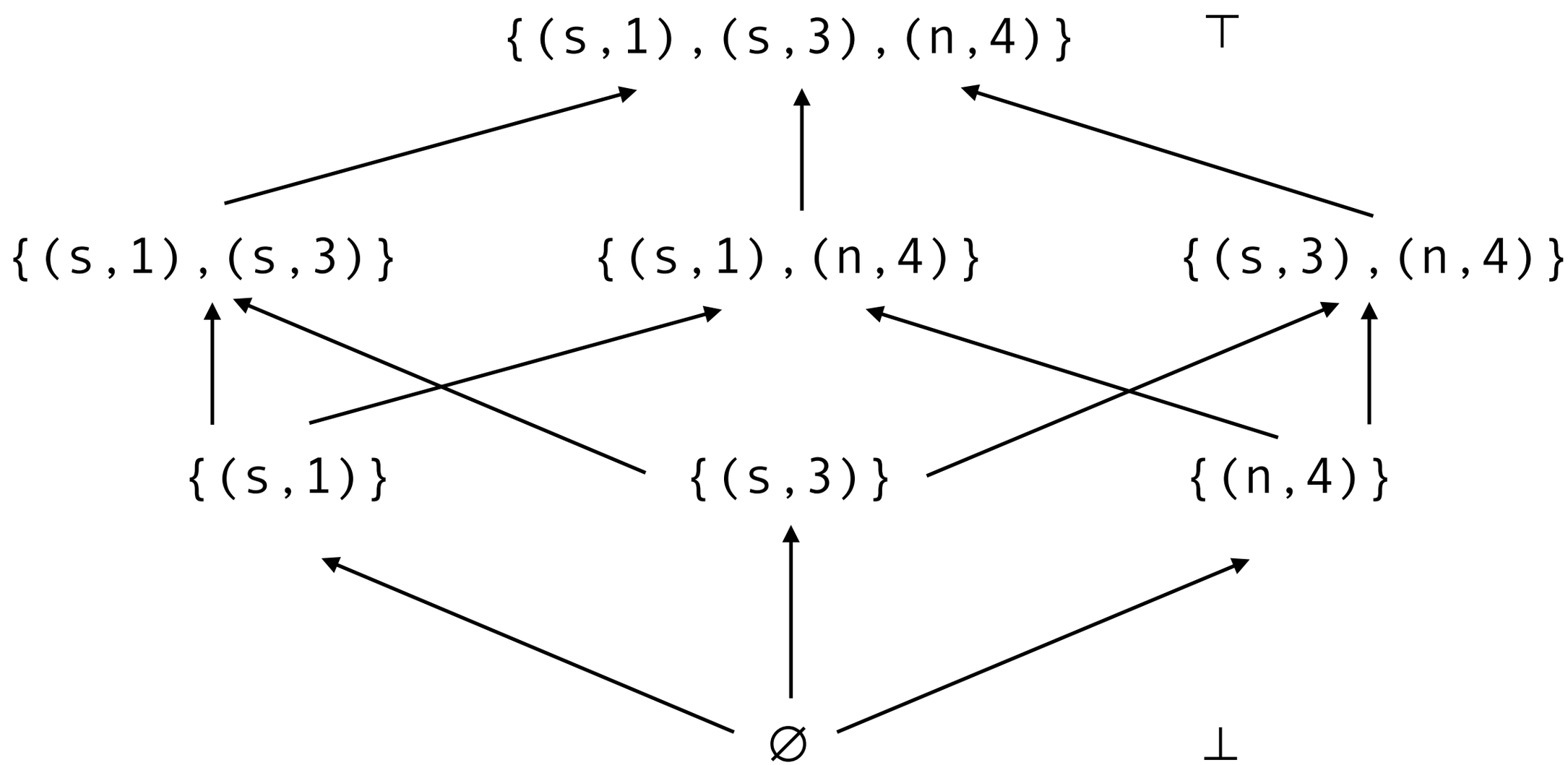
$RD_{entry}(1) = RD_{exit}(0)$
$RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)$
$RD_{entry}(3) = RD_{exit}(2)$
$RD_{entry}(4) = RD_{exit}(3)$
$RD_{entry}(5) = RD_{exit}(2)$

$RD_{exit}(0) = \{(n,0)\}$
$RD_{exit}(1) = RD_{entry}(1) \backslash all(s) \cup \{(s,1)\}$
$RD_{exit}(2) = RD_{entry}(2)$
$RD_{exit}(3) = RD_{entry}(3) \backslash all(s) \cup \{(s,3)\}$
$RD_{exit}(4) = RD_{entry}(4) \backslash all(n) \cup \{(n,4)\}$
$RD_{exit}(5) = RD_{entry}(5)$



# *Lattices*

- Facts range over ***lattices***: partial orders with ***joins*** (least upper bounds) and ***meets*** (greatest lower bounds).



# *Lattices*

- Facts range over ***lattices***: partial orders with ***joins*** (least upper bounds) and ***meets*** (greatest lower bounds).

- A partial order is a set X and an ordering $(X, \sqsubseteq)$ that is:
  - Reflexive; $\forall x.\ x \sqsubseteq x$
  - Transitive; $\forall x,y,z.\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Anti-symmetric; $\forall x,y.\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

- Lattices must also have unique joins and meets for any 2 points. *Complete lattices* have unique joins/meets for any set.

- Cartesian product of lattices is a lattice. Map with a lattice co-domain is a lattice.

# *Reaching definitions* analysis

- The 11 sets $RD_{exit}(0)$, $RD_{entry}(1)$, $RD_{exit}(1)$,…$RD_{exit}(5)$, are defined in terms of one another. Written as a vector of sets:

$$\overrightarrow{RD}$$

- Our set of equations can be turned into a ***monotonic*** F:

$$\overrightarrow{RD_0} \sqsubseteq \overrightarrow{RD_1} \Rightarrow F(\overrightarrow{RD_0}) \sqsubseteq F(\overrightarrow{RD_1})$$

- So that a satisfying vector of reachable defs is a ***fixed point:***

$$\overrightarrow{RD} = F(\overrightarrow{RD}) = F^n(\perp) \quad \textit{for some } n$$

- For example, the join point would end up encoded as:

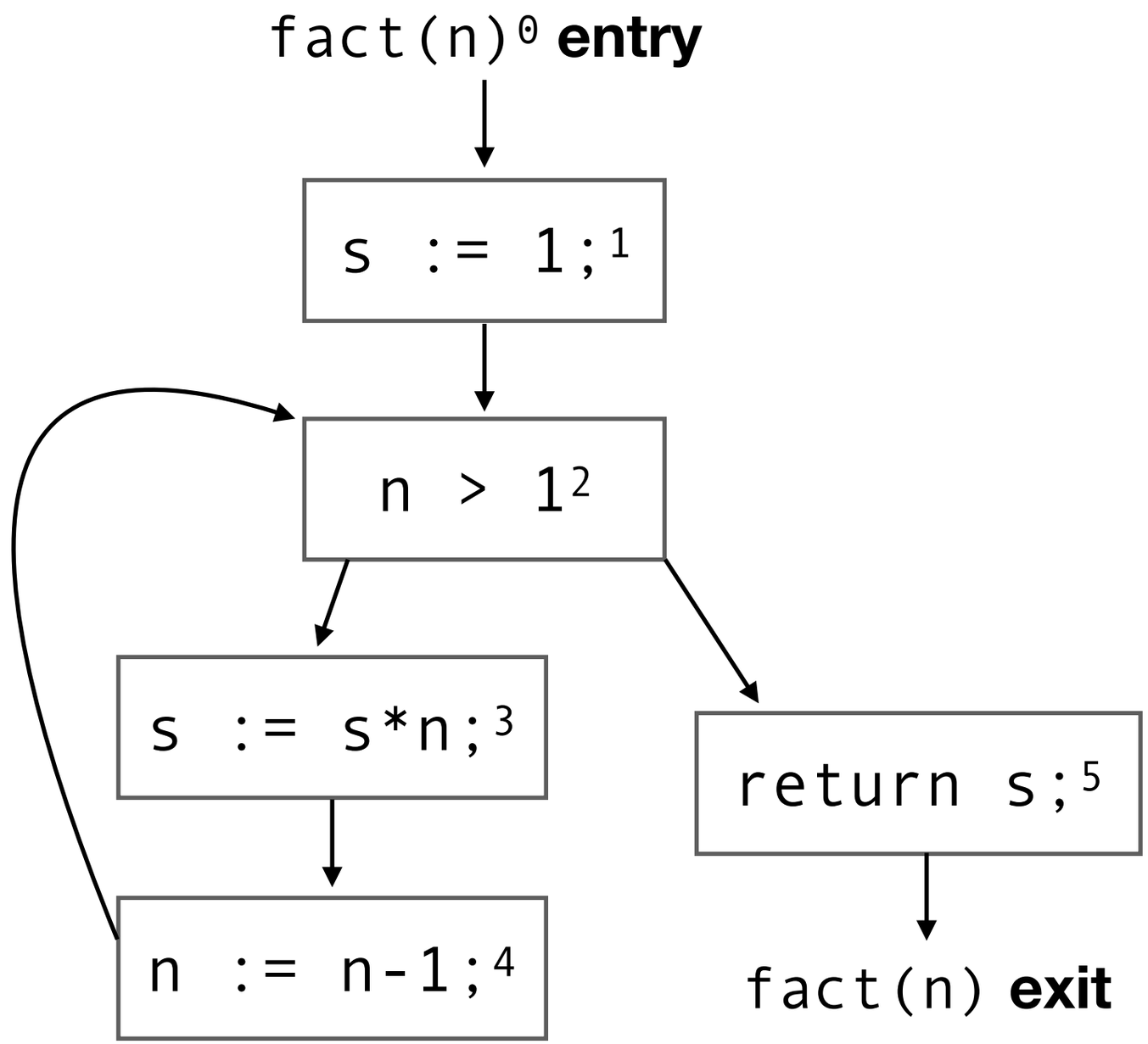
$$F(\ldots, RD_{exit}(1), \ldots, RD_{exit}(4), \ldots) = (\ldots, RD_{exit}(1) \cup RD_{exit}(4), \ldots)$$

# *Very busy expressions* analysis

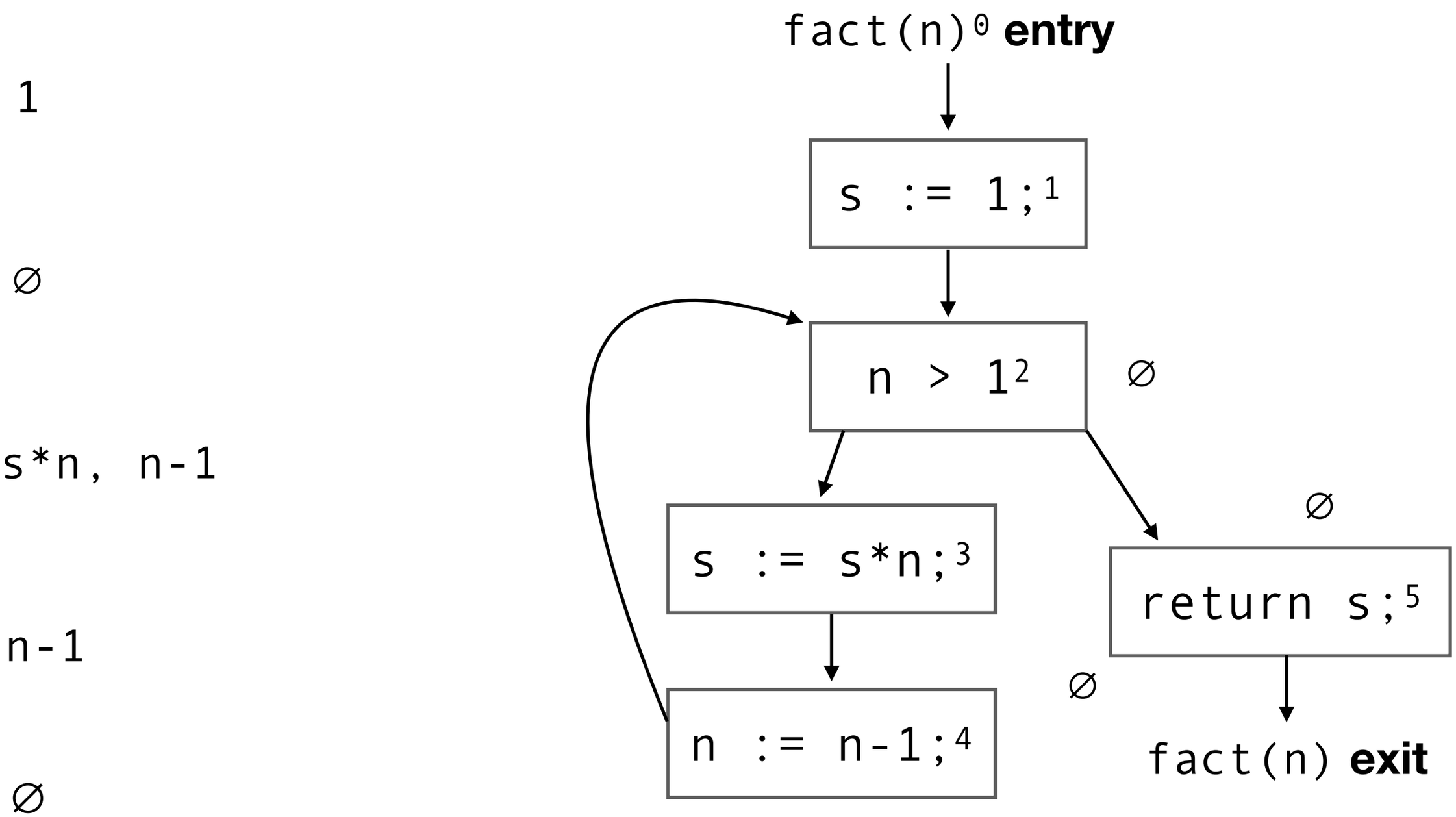
- Computes a set of expressions that are computed down all paths forward before any subexpressions change value.
- Assignments represent GEN for the right hand side and KILL for expressions containing the right hand side (assigned var).
- Is a ***backward must*** data-flow analysis:
    - Propagates a set of computed expressions *backward*.
    - Computes the ***meet*** (GLB, intersection) of entry(s') in s'∈succ(s) at each fork point to obtain exit(s).

# *Very busy expressions* analysis

| Stmt | GEN | KILL |
|---|---|---|
| `s := 1;`$^{1}$ | 1 | s<br>s*n |
| `n > 1`$^{2}$ | | |
| `s := s*n;`$^{3}$ | s*n | s<br>s*n |
| `n := n-1;`$^{4}$ | n-1 | n-1<br>s*n |
| `return s;`$^{5}$ | | |



# *Very busy expressions* analysis



# May/Must & Forward/Backward

| | May | Must |
|---|---|---|
| Forward | Forward, computes exit(s) from entry(s)<br>Join (∪) at CFG join points<br>e.g., Reaching Defs (use-def)<br>(which assignments reach uses) | Forward, computes exit(s) from entry(s)<br>Meet (∩) at CFG join points<br>e.g., Available Expressions |
| Backward | Backward, computes entry(s) from exit(s)<br>Join (∪) at CFG fork points<br>e.g., Live Variables | Backward, computes entry(s) from exit(s)<br>Meet (∩) at CFG fork points<br>e.g., Very Busy Expressions |

```
exit(s) = ⊥
W = all statements s  //worklist
while W not empty:
   remove s from W

   entry(s) = ∪_{s'∈pred(s)} exit(s')

   update = (entry(s) \ kill(s)) ∪ gen(s)
   if update != exit(s):
      exit(s) = update
      W = W ∪ succ(s)
```

*Forward may* analysis

```
exit(s) = ⊤ // except ⊥ at function entry
W = all statements s  //worklist
while W not empty:
   remove s from W
   entry(s) = ⋂_{s'∈pred(s)} exit(s')
   update = (entry(s) \ kill(s)) ∪ gen(s)
   if update != exit(s):
      exit(s) = update
      W = W ∪ succ(s)
```

*Forward must* analysis

```
entry(s) = ⊥
W = all statements s  //worklist
while W not empty:
   remove s from W

   exit(s) = ∪_{s'∈succ(s)} entry(s')

   update = (exit(s) \ kill(s)) ∪ gen(s)
   if update != entry(s):
      entry(s) = update
      W = W ∪ succ(s)
```

*Backward may* analysis

```
entry(s) = ⊤ // except ⊥ at function exit
W = all statements s  //worklist
while W not empty:
   remove s from W
   exit(s) = ⋂_{s'∈succ(s)} entry(s')
   update = (exit(s) \ kill(s)) ∪ gen(s)
   if update != entry(s):
      entry(s) = update
      W = W ∪ succ(s)
```

# Back*ward must* analysis

# Abstract interpretation

- A general methodology for ***justifying*** or ***calculating*** sound analyses, given a precise semantics for the target language.

- Abstract interpretation establishes ***abstract*** semantic domains and a ***Galois connection*** between concrete and abstract.

- A function alpha ($\alpha: X \to \hat{X}$) defines a notion of abstraction.

- A function gamma ($\gamma: \hat{X} \to X$) defines a corresponding notion of concretization ($\alpha$ implies $\gamma$ and vice versa; more on this…).

- A concrete interpreter ($F: X \to X$) and Galois connection can be used to justify or calculate an abstract interpretation:

$$\alpha \circ F \circ \gamma \sqsubseteq \hat{F}$$

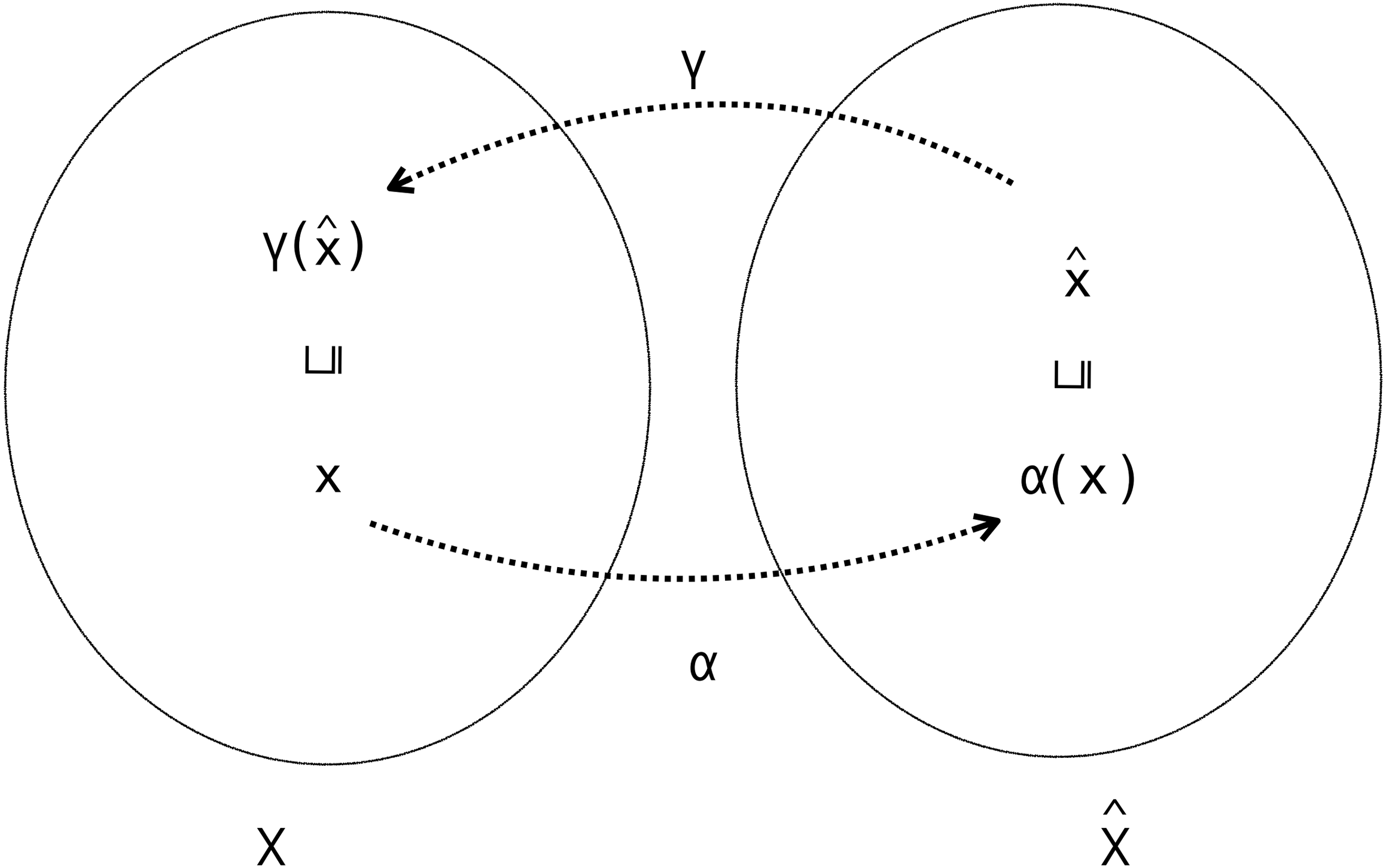
# Abstraction/Concretization (*Galois*)

$$\alpha(x) \sqsubseteq \hat{x}$$

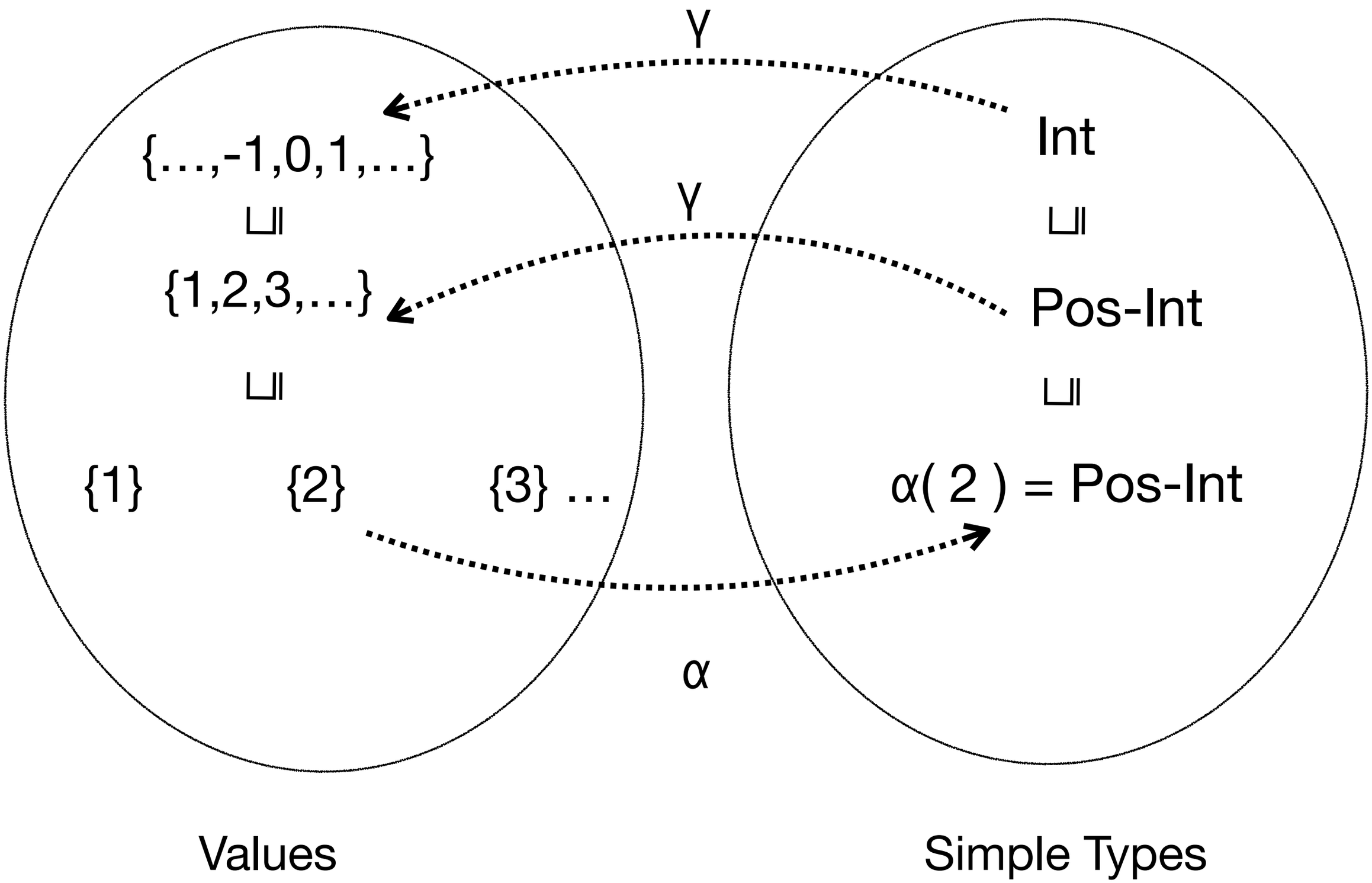
if and only if

$$x \sqsubseteq \gamma(\hat{x})$$

# Abstraction/Concretization (*Galois conn.*)

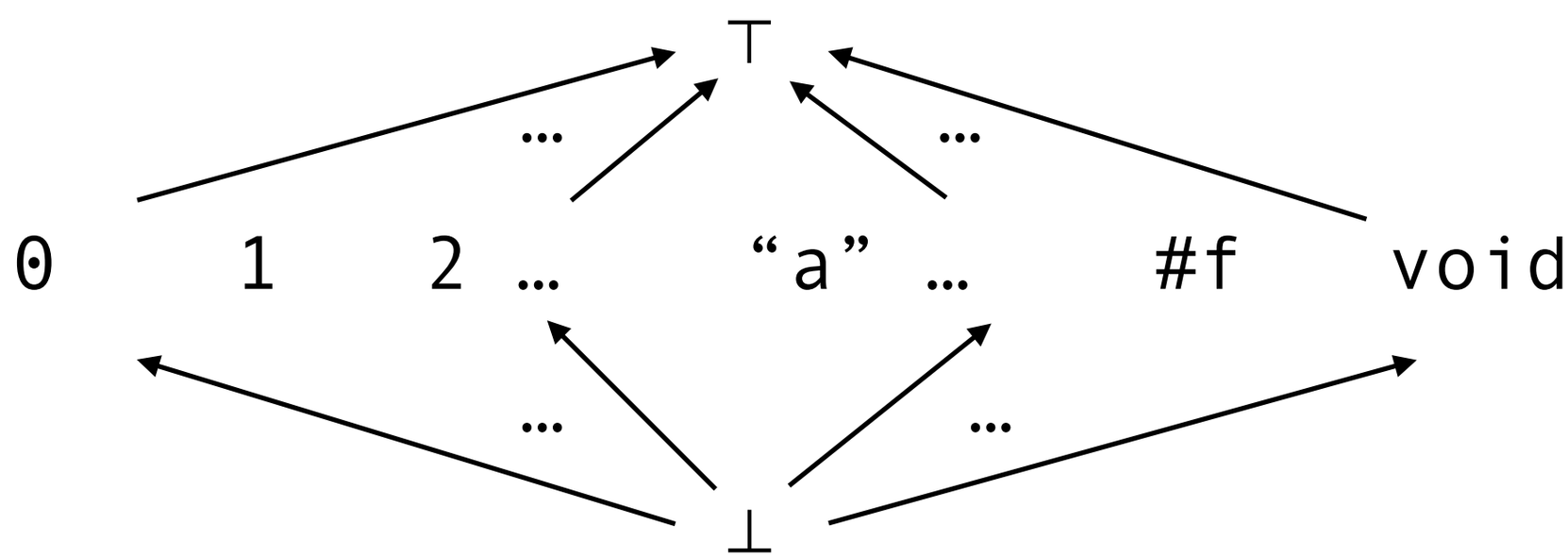


# Abstraction/Concretization (*Galois conn.*)



# *Constant Propagation*

- Forward must style of DFA. Or as an abstract interpretation:
- Uses a flat lattice of constants with top and bottom (C, ⊑):




- Facts become sets of pairs (Var x C) or a map (Env: Var → C).

# Flow analysis

- Intraprocedural analysis: considers functions independently.
- Interprocedural analysis: considers multiple functions together.
- Whole-program analysis: considers an entire program at once.
- DFA is great for simple, local-variable-focused analyses.
- Analysis of heap-allocated data is *much* harder.
- The simple case is called *pointer analysis* (aliases, nullable).
- The general case is called *shape analysis* (full data-structures).

# What about Scheme or ANF/CPS IRs?

**Depends on call sites for the lambda that binds it.**

```
(lambda (k f x y)
  (let ([a (prim + x y)])
    (f k a)))
```

**What value can `f` be?**

Data-flow depends upon control-flow.

**Depends on the possible values of parameter `f`.**

```
(lambda (k f x y)
  (let ([a (prim + x y)])
    (f k a)))
```

**Where does control propagate from this call-site?**

Control-flow depends upon data-flow.

The *higher-order control-flow problem*:
Data-flow and control-flow properties are thoroughly entangled and mutually dependent.

The solution?

*Control-flow analysis*:
Simultaneously model control-flow behavior and data-flow behavior in a single analysis.

# Control-flow analysis

- Use abstract interpretation to produce a single uniform analysis of all interdependent program properties.
- CFAs are whole-program interprocedural analyses. (Shivers 1991)
- ***k***-CFA tracks k latest call-sites; bindings have context. (EXPTIME)
- 0-CFA produces a single model of each function. ($O(n^3)$)
- ***Abstracting abstract machines (AAM):*** a unified methodology for deriving CFAs from concrete (precise) abstract machines! (Van Horn and Might 2010)

# CPS lambda calculus

```
 e∈Exp  ::= (ae ae …)
ae∈AExp ::= x | lam
lam∈Lam ::= (lambda (x …) e)
```

x∈Var is a set of variables

# CPS lambda calculus (semantics)

**DOMAINS**

State: Exp x Env
Env: Var → Clo
Clo: Lam x Env

**ATOMIC EVAL**

A(lam,env) = (lam,env)
A(x,env) = env(x)

**SMALL-STEP TRANSITION**

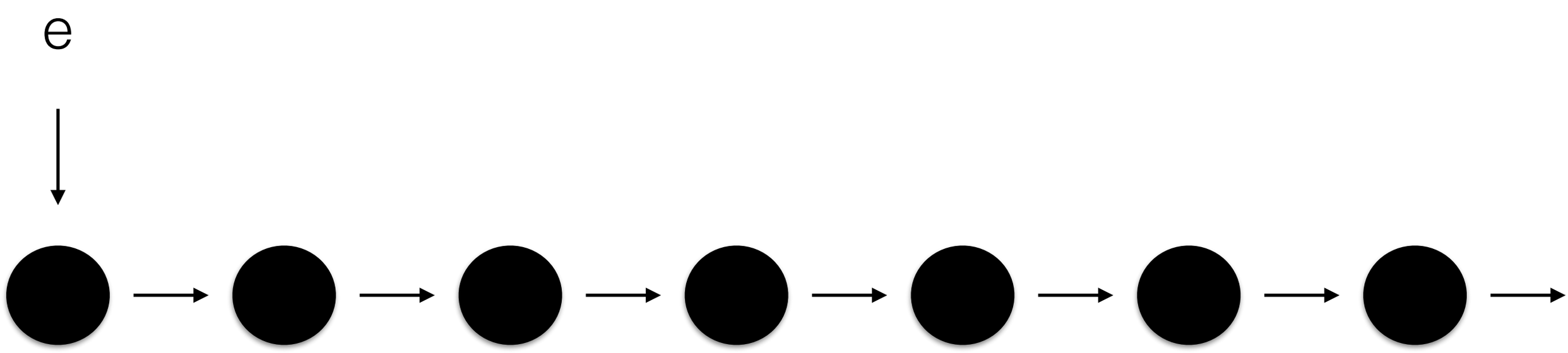
$((ae_0 \ldots ae_j), env) \rightarrow (e_0, env')$, *where*

$((lambda\ (x_1 \ldots x_j)\ e_0), env_c) = A(ae_0)$

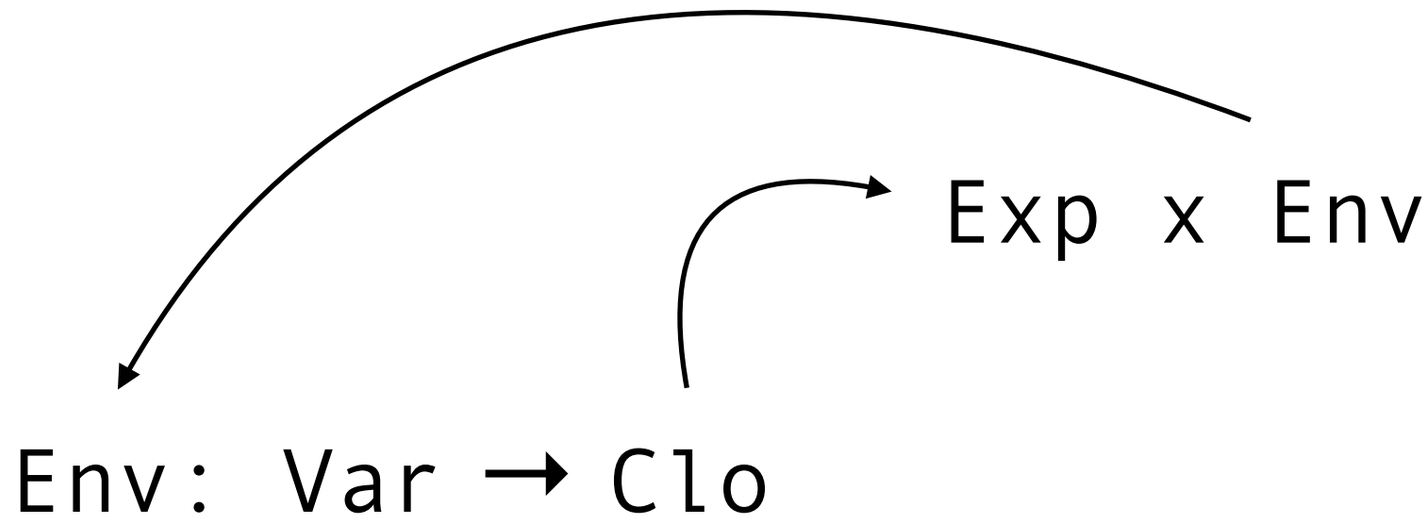
$clo_i = A(ae_i)$

$env' = env_c[x_i \rightarrow clo_i]$

# Collecting semantics (generates a trace)




Tarski. (1955)




Might. Abstract interpreters for free. (2010)

# CPS store-passing semantics

**DOMAINS**

```
State: Exp x Env X Store
  Env: Var → Addr
Store: Addr → Clo
  Clo: Lam x Env
```

`Addr:` some infinite set

**ATOMIC EVAL**

```
A(lam,env,st) = (lam,env)
  A(x,env,st) = st(env(x))
```

# CPS store-passing semantics

**SMALL-STEP TRANSITION**

$$((ae_0 \ldots ae_j), env, st) \rightarrow (e_0, env', st'),$$

*where* $((\texttt{lambda}\ (x_1 \ldots x_j)\ e_0), env_c) = A(ae_0)$

$$clo_i = A(ae_i)$$

$$env' = env_c[x_i \rightarrow alloc(x_i)]$$

$$st' = st[alloc(x_i) \rightarrow clo_i]$$

$alloc(x_i)$ = fresh address

= $x_i$ (yields 0-CFA)

# Now we may finitize the set Addr

$$\mathtt{State}: \mathtt{Exp} \times \mathtt{Env} \times \mathtt{Store}$$

$$\mathtt{Env}: \mathtt{Var} \to \widehat{\mathtt{Addr}}$$

$$\mathtt{Store}: \widehat{\mathtt{Addr}} \to \mathtt{Clo}$$

$$\mathtt{Clo}: \mathtt{Lam} \times \mathtt{Env}$$

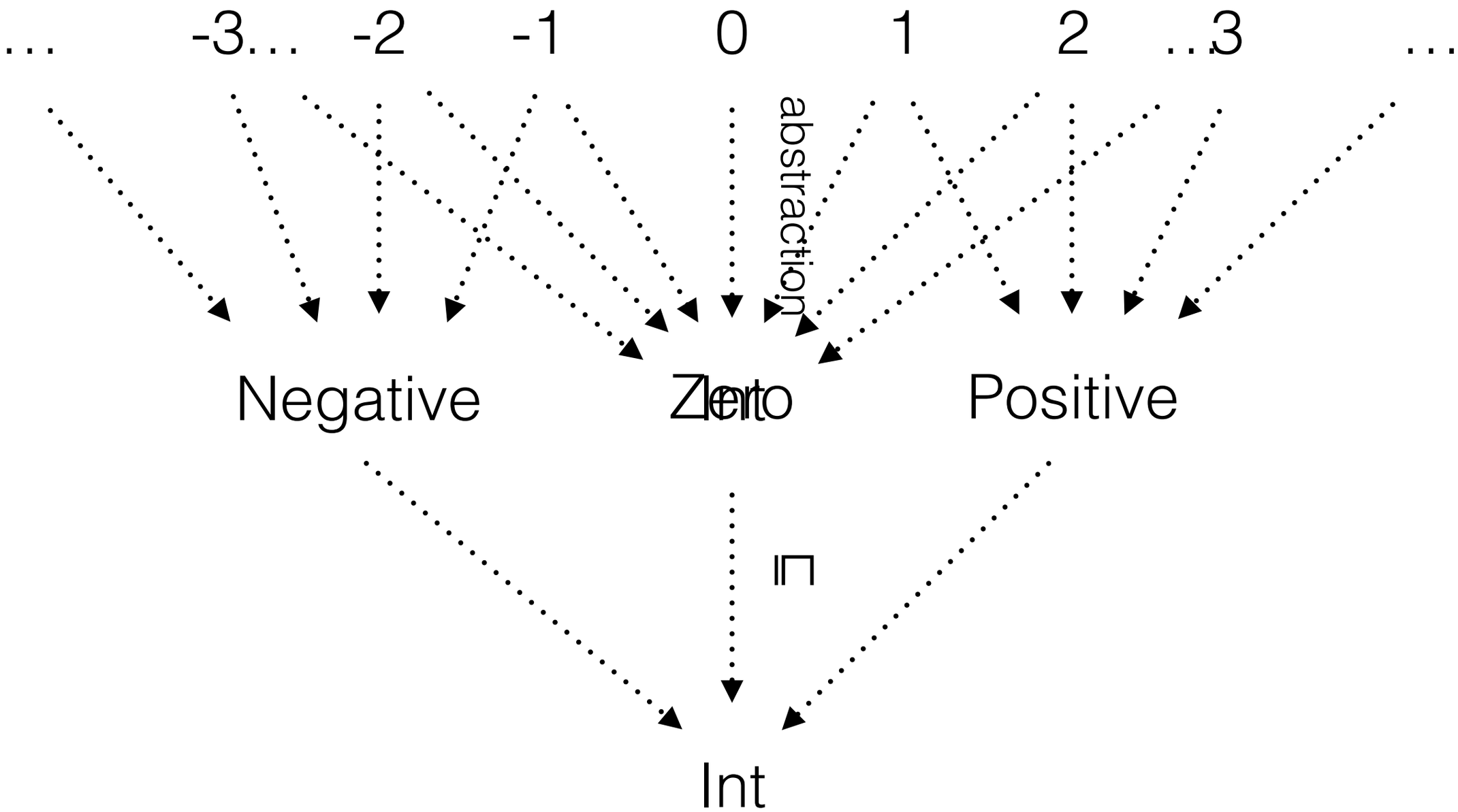
# Now we may finitize the set Addr

$$\widehat{State} : Exp \times \widehat{Env} \times \widehat{Store}$$

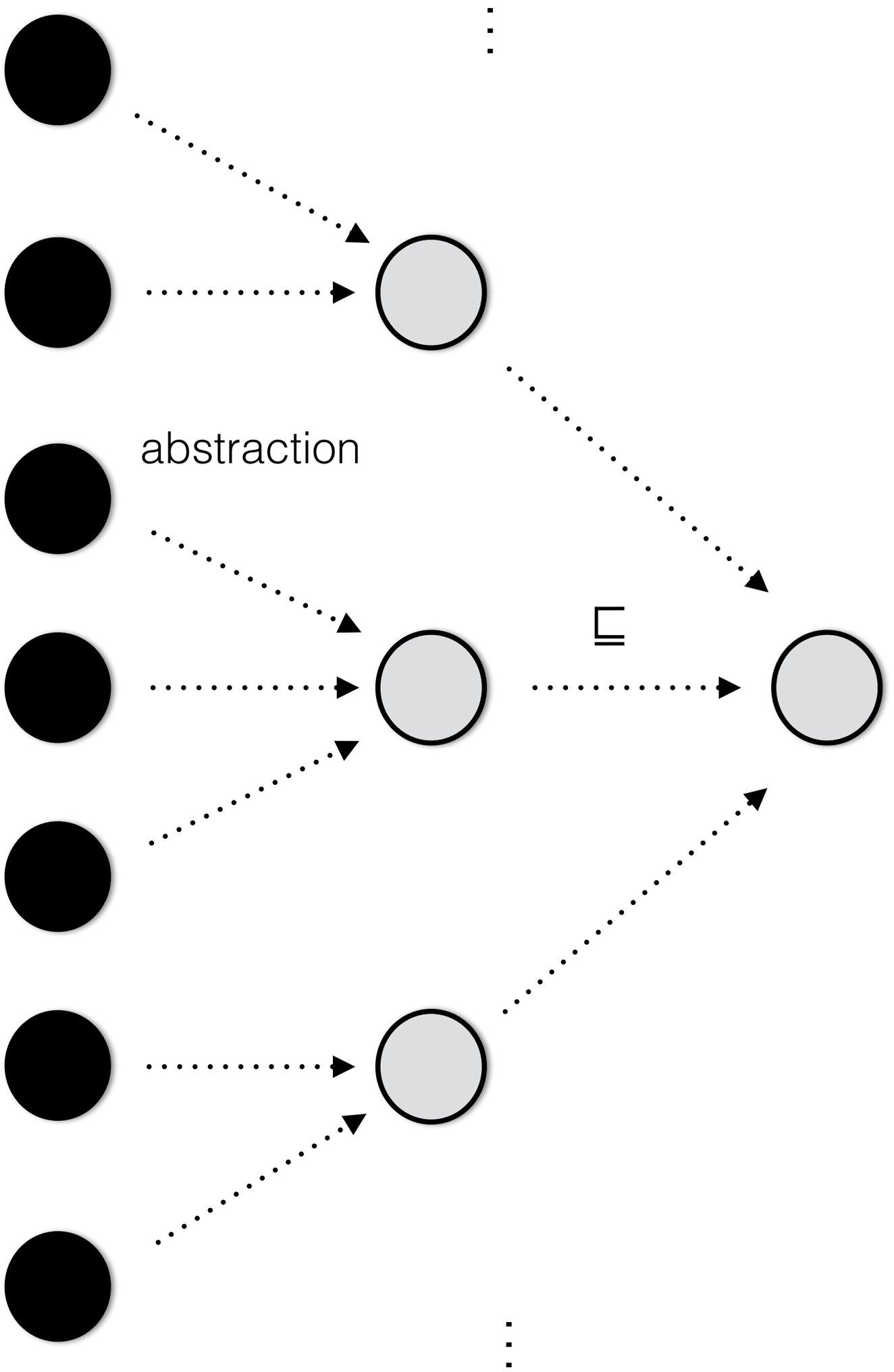
$$\widehat{Env} : Var \to \widehat{Addr}$$

$$\widehat{Store} : \widehat{Addr} \to \mathcal{P}(\widehat{Clo})$$

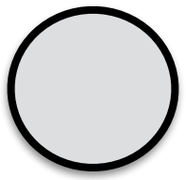
$$\widehat{Clo} : Lam \times \widehat{Env}$$




Cousot and Cousot (1977), Cousot (2000)

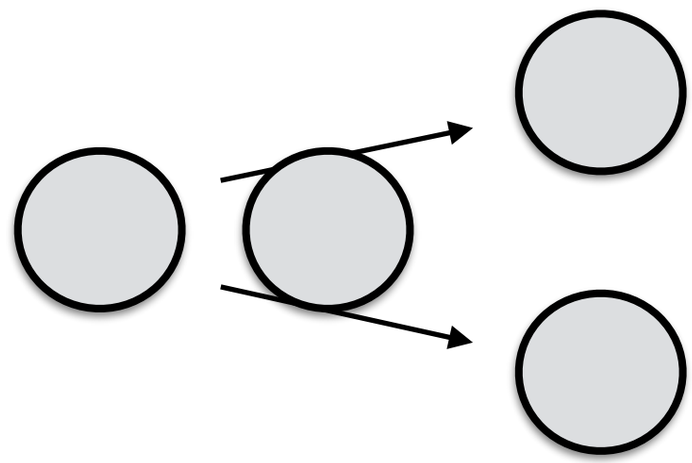

abstraction
⊑

# Abstract abstract machines

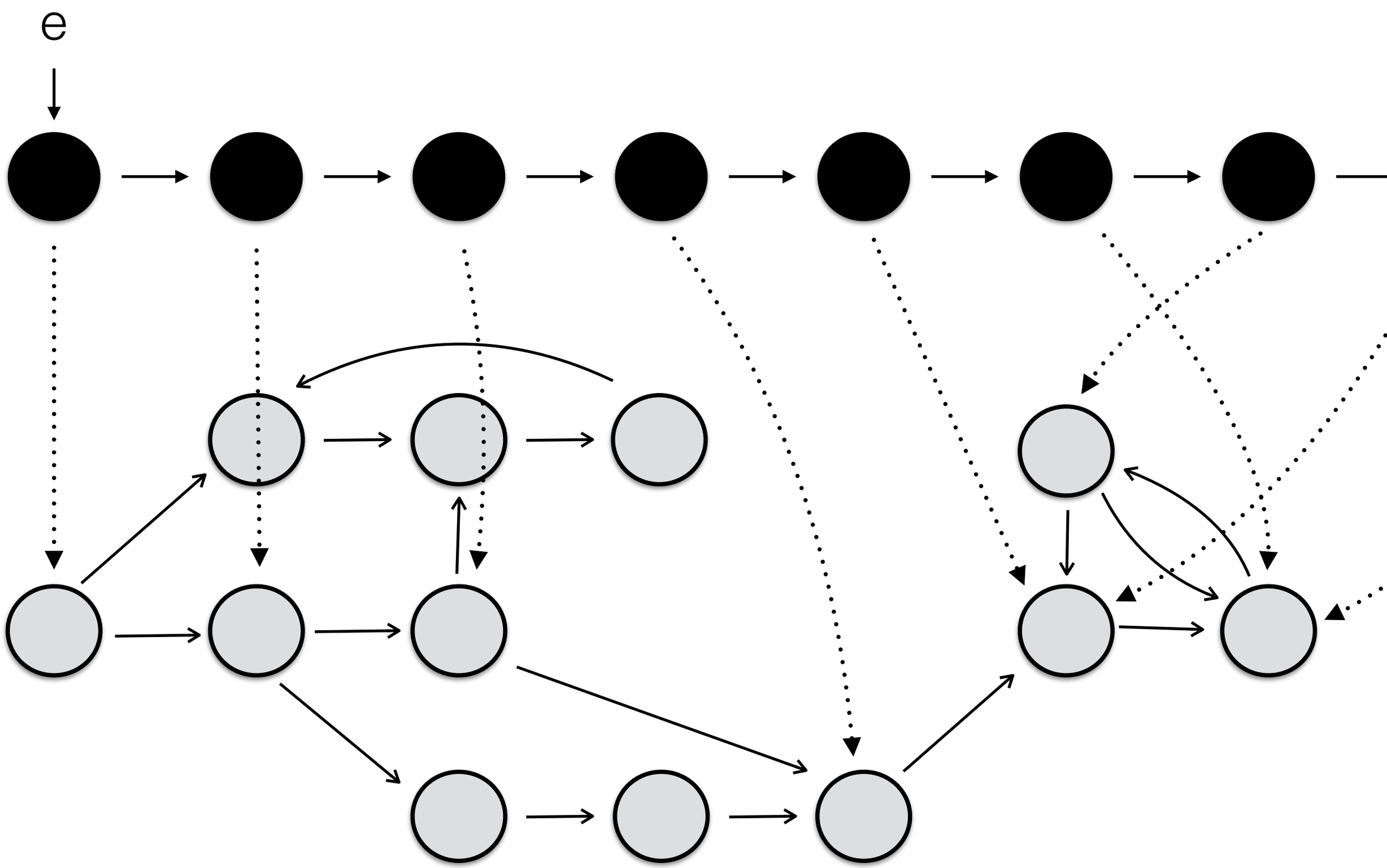


$$\left( \texttt{(f x)} \;,\; \begin{array}{l} [x \rightarrow a_x, \\ \;y \rightarrow a_y, \\ \;f \rightarrow a_f] \end{array} \;,\; \begin{array}{l} [a_x \rightarrow \{\mathrm{Int}\}, \\ \;a_y \rightarrow \{\mathrm{Int}\}, \\ \;a_f \rightarrow \{\texttt{(}\lambda\ \texttt{(w)}\ \ e_1\texttt{)}, \\ \qquad\quad \texttt{(}\lambda\ \texttt{(z)}\ \ e_2\texttt{)}\}] \end{array} \right)$$

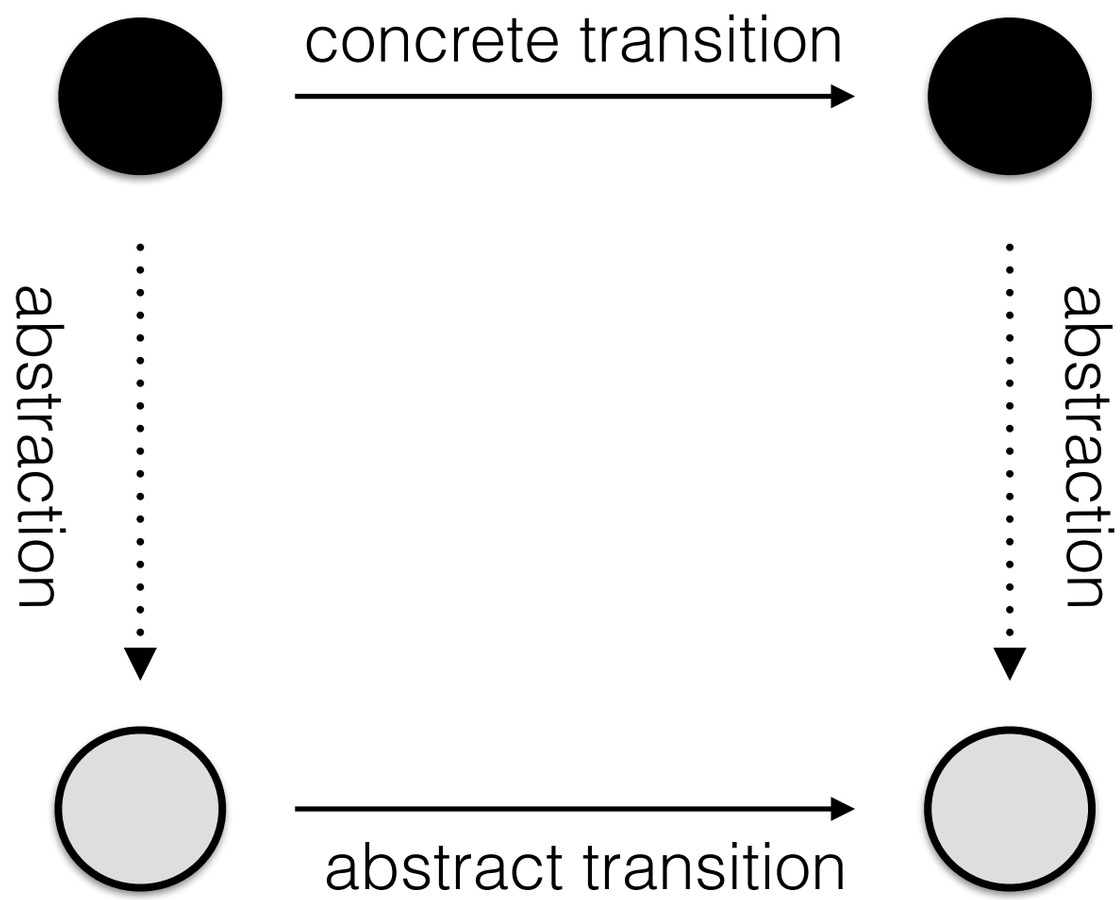
# Abstract-state transition



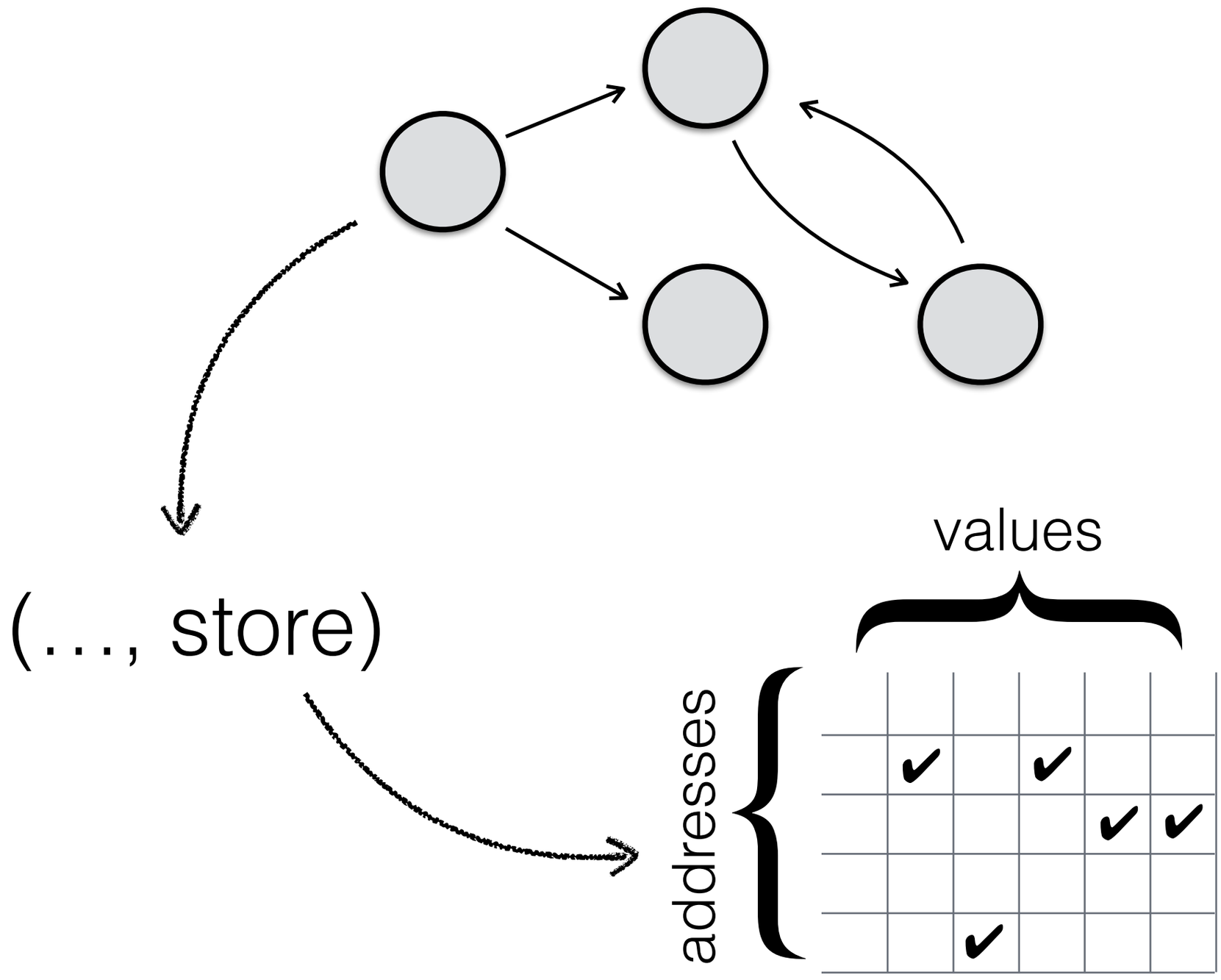
$$\left( \texttt{(f x)} \;,\; \begin{array}{l} [x \rightarrow a_x, \\ \;\; y \rightarrow a_y, \\ \;\; f \rightarrow a_f] \end{array} \;,\; \begin{array}{l} [a_x \rightarrow \{\mathrm{Int}\}, \\ \;\; a_y \rightarrow \{\mathrm{Int}\}, \\ \;\; a_f \rightarrow \{\texttt{(}\lambda\ \texttt{(w)}\ \ e_1\texttt{)}, \\ \qquad\qquad \texttt{(}\lambda\ \texttt{(z)}\ \ e_2\texttt{)}\}] \end{array} \right)$$


e

# Soundness

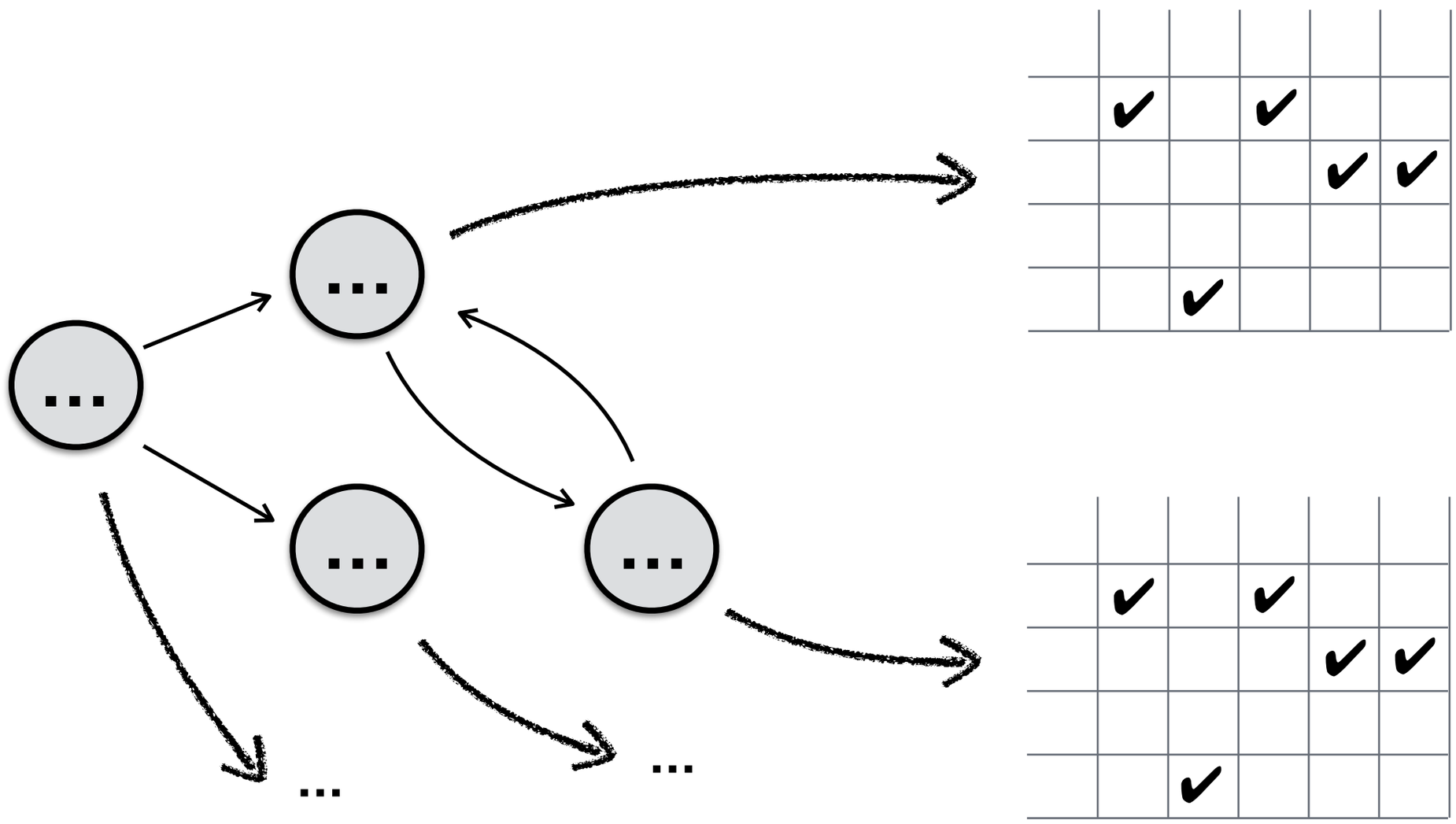

concrete transition
abstraction
abstraction
abstract transition

# Exponential complexity



# Global store widening

(*flow-sensitive*)

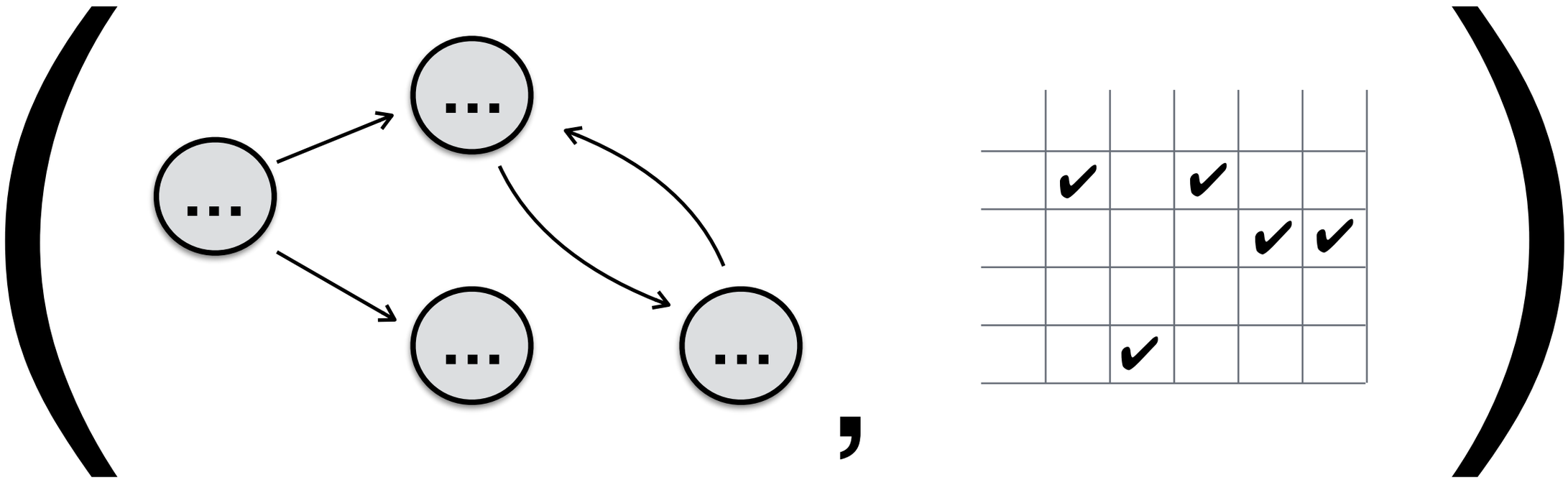



Control-flow graph

per-point stores

# Global store widening

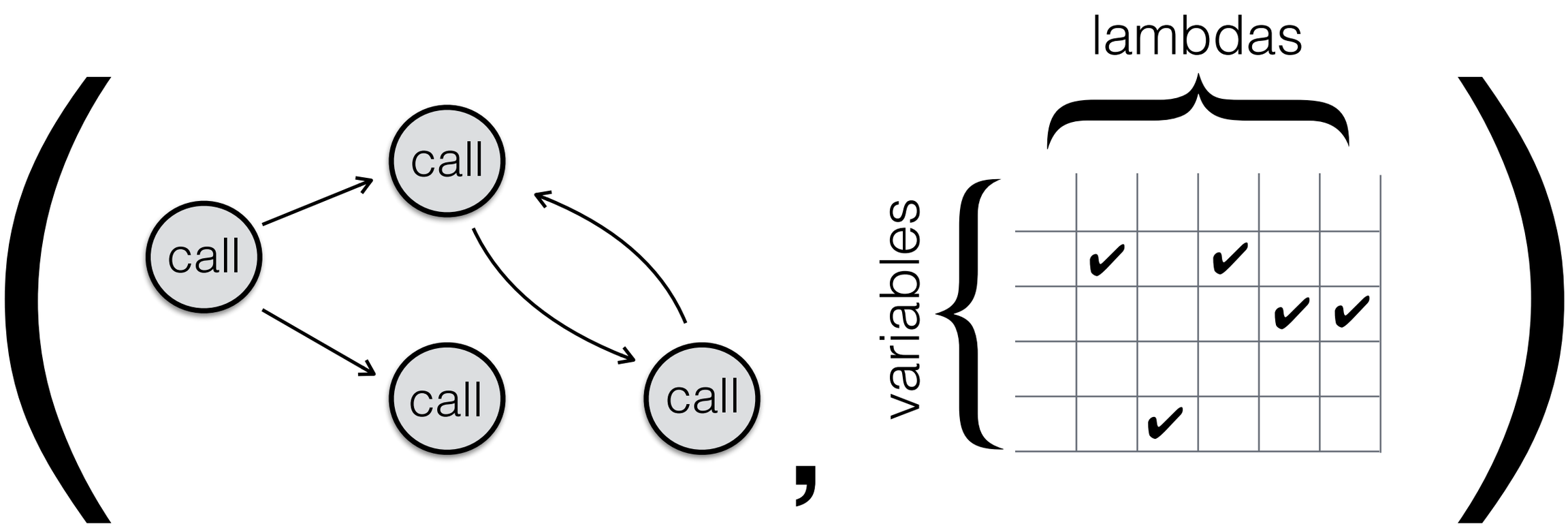
(*flow-insensitive*)




Control-flow graph

Global store

# Flow-insensitive (CPS) 0-CFA




Control-flow graph (of call-sites)

Variables map to sets of lambdas

Let’s live-code this 0-CFA