软件分析与架构设计

数据流分析

何冬杰

重庆大学

回顾收集语义

The **collecting semantics** is used to define a collection of all the states (defined by operational semantics) of the program at a given point.

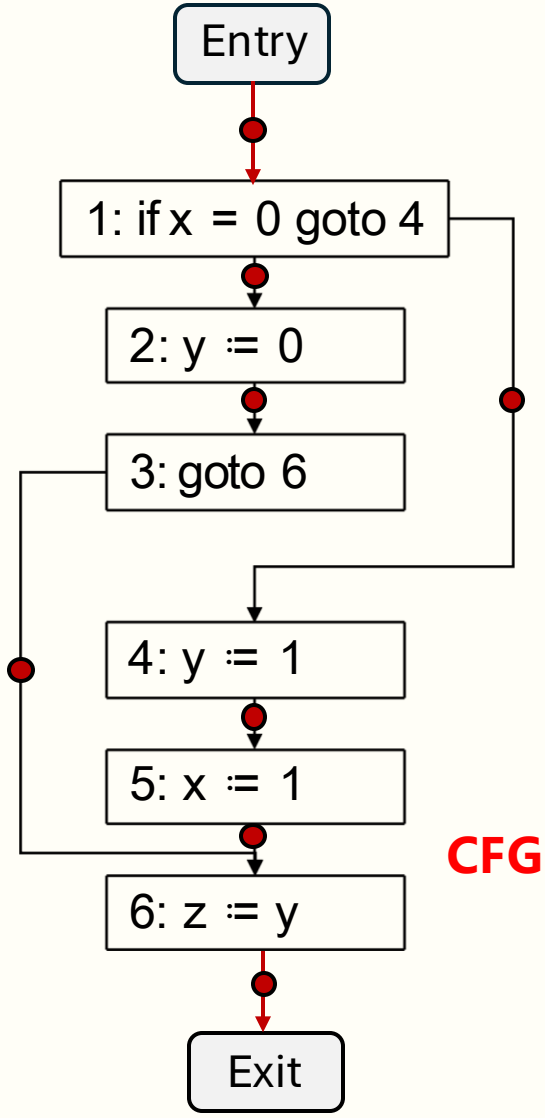
- **指针分析和控制流分析均可看作是AST树上的收集语义分析**
  - Collecting program state for AST nodes without considering execution orders
  - Collected states are considered to be **valid at all program points** rather than at specific program points
  - Relatively **efficient** in both time and memory, but **less precise**
- **数据流分析可看作是在CFG上的收集语义分析**
  - Computes universal properties about the program state at specific program points
  - flow sensitive, i.e., considering execution orders

Review: 控制流图（Control Flow Graph）

- **控制流图：表示语句执行顺序的图**
  - **Nodes**：statements
  - **Edges**：(s1, s2) is an edge iff s1 and s2 can be executed consecutively aka “control flow”
- **Properties:**
  - Only one **entry** node
  - Only one **exit** (terminal) node
  - Weakly connected
    - 边表示控制流的可能转移关系
- **Program Points:** •

$$
\begin{aligned}
1:&\quad \text{if } x = 0 \text{ goto } 4\\
2:&\quad y := 0\\
3:&\quad \text{goto } 6\\
4:&\quad y := 1\\
5:&\quad x := 1\\
6:&\quad z := y
\end{aligned}
$$

三地址码



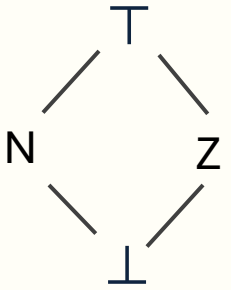
CFG

Revisit Zero Analysis

- **The domain value of each variable in concrete analysis could be infinite**

```
1. Read x
2. y = 2 * x
3. z = 0
4. if y > 0 goto 6
5. z = 2 * x - y
6. ...
```

$$x \mapsto \{-\infty, \cdots, -2, -1, 0, 1, 2, \cdots + \infty\}$$



- **Abstraction:**
  - 抽象域（abstract domain） $\boldsymbol{L} = \{\bot, \mathrm{Z}, \mathrm{N}, \top\}$ forms a lattice
  - $\boldsymbol{\alpha}: \mathbb{Z} \to \boldsymbol{L}$: maps value in concrete domain into abstract domain
    - $\alpha(0) = \mathrm{Z};\ \alpha(\mathrm{n}) = \mathrm{N}\ \text{if}\ \mathrm{n} \neq 0$
  - **Abstract state** $\boldsymbol{\sigma}: \mathrm{Var} \to \boldsymbol{L}$
  - **flow function:** $\boldsymbol{f_Z}: \boldsymbol{\sigma} \times \boldsymbol{Inst} \to \boldsymbol{\sigma'}$
    - transform the **input state** of an instruction to the **output state**
    - modeling the *abstract operational semantics*

Zero Analysis: Flow Insensitively

$$\textbf{\textit{Inst}} ::= x := n \mid x := y \mid x := y\ op\ z \mid \text{goto } n \mid \text{if } x\ \text{op}_r\ 0 \text{ goto } n$$

❑Flow functions:

$$f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z \sqcup \sigma(x)]$$

$$f_Z[\![x := n]\!](\sigma) = \sigma[x \mapsto N \sqcup \sigma(x)] \ \ where\ n \neq 0$$

$$f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y) \sqcup \sigma(x)$$

$$f_Z[\![x := y\ op\ z]\!](\sigma) = \sigma[x \mapsto \top]$$

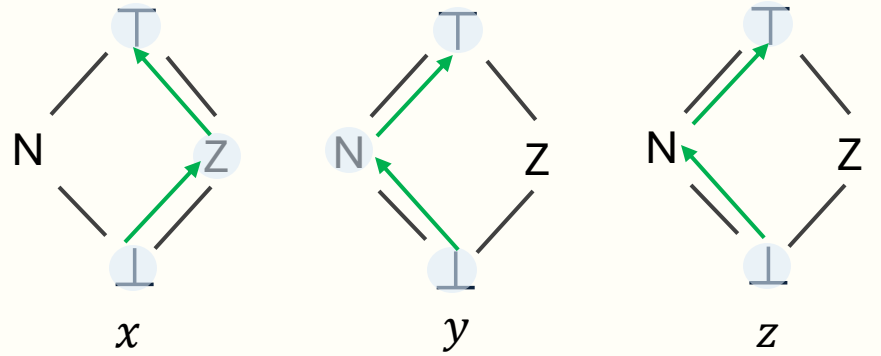
$$f_Z[\![\text{goto } n]\!](\sigma) = \sigma$$

$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma$$

❑Example: Straightline Code

➢For flow-insensitive analysis, we only have one global state

$$\begin{aligned}
1: &\quad x := 0\\
2: &\quad y := 1\\
3: &\quad z := y\\
4: &\quad y := z + x\\
5: &\quad x := y - z
\end{aligned}$$



Useless results!
Could be more precise?

Zero Analysis: Flow Sensitively

❑Flow functions I:

$\textit{Inst} ::= x := n \mid x := y \mid x := y \; op \; z \mid \text{goto } n \mid \text{if } x \; \text{op}_r \; 0 \text{ goto } n$

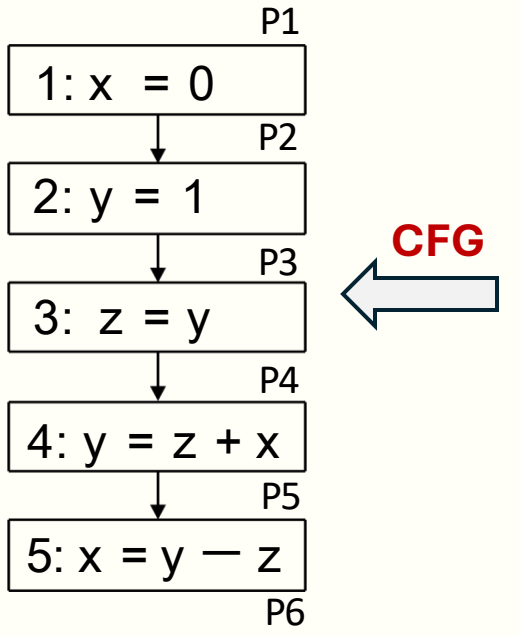
$$f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![x := n]\!](\sigma) = \sigma[x \mapsto N] \text{ where } n \neq 0$$
$$f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$$
$$f_Z[\![x := y \; op \; z]\!](\sigma) = \sigma[x \mapsto \top]$$
$$f_Z[\![\text{goto } n]\!](\sigma) = \sigma$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma$$

❑Example: Straightline Code

	x	y	z
P1	⊥	⊥	⊥
P2	Z	⊥	⊥
P3	Z	N	⊥
P4	Z	N	N
P5	Z	⊤	N
P6	⊤	⊤	N

DFA

- differ at different program points
- more precise!
- could be more precise?



CFG

$$\begin{aligned} 1: &\quad x := 0 \\ 2: &\quad y := 1 \\ 3: &\quad z := y \\ 4: &\quad y := z + x \\ 5: &\quad x := y - z \end{aligned}$$

Zero Analysis: Flow Sensitively

❑Flow functions II:

$$f_Z[\![x := y - y]\!](\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![x := y + z]\!](\sigma) = \sigma[x \mapsto \sigma(y)] \quad \text{where } \sigma(z) = Z$$

specialize for precision

$$f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![x := n]\!](\sigma) = \sigma[x \mapsto N] \quad \text{where } n \neq 0$$
$$f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$$

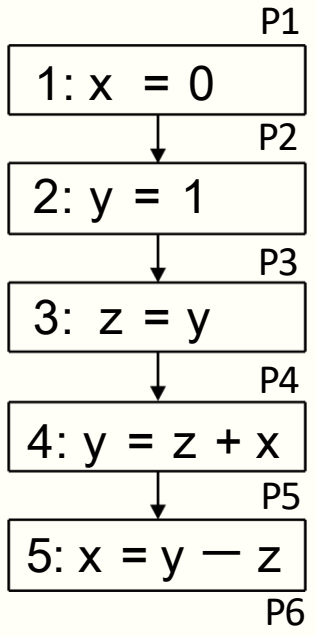
$$f_Z[\![x := y \ op \ z]\!](\sigma) = \sigma[x \mapsto \top]$$
$$f_Z[\![\text{goto } n]\!](\sigma) = \sigma$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma$$

❑Example: Straightline Code

	x	y	z
P1	⊥	⊥	⊥
P2	Z	⊥	⊥
P3	Z	N	⊥
P4	Z	N	N
P5	Z	N	N
P6	⊤	N	N

DFA

- differ at different program points
- more precise!
- could be more precise?



CFG

$$1: \quad x := 0$$
$$2: \quad y := 1$$
$$3: \quad z := y$$
$$4: \quad y := z + x$$
$$5: \quad x := y - z$$

Zero Analysis: Flow Sensitively

❑Flow functions III:

$$f_Z[\![x := y - y]\!](\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![x := y + z]\!](\sigma) = \sigma[x \mapsto \sigma(y)] \quad \text{where } \sigma(z) = Z$$

specialize for precision

$$f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![x := n]\!](\sigma) = \sigma[x \mapsto N] \quad \text{where } n \neq 0$$
$$f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$$

$$f_Z[\![x := y \ op \ z]\!](\sigma) = \sigma[x \mapsto \top]$$
$$f_Z[\![\text{goto } n]\!](\sigma) = \sigma$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) = \sigma$$

specialize for precision

$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) = \sigma[x \mapsto Z]$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) = \sigma[x \mapsto N]$$

Note: flow functions here are functions between a pair of program points, rather than on an instruction.

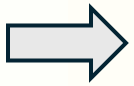
❑One can design other specialisations for precision

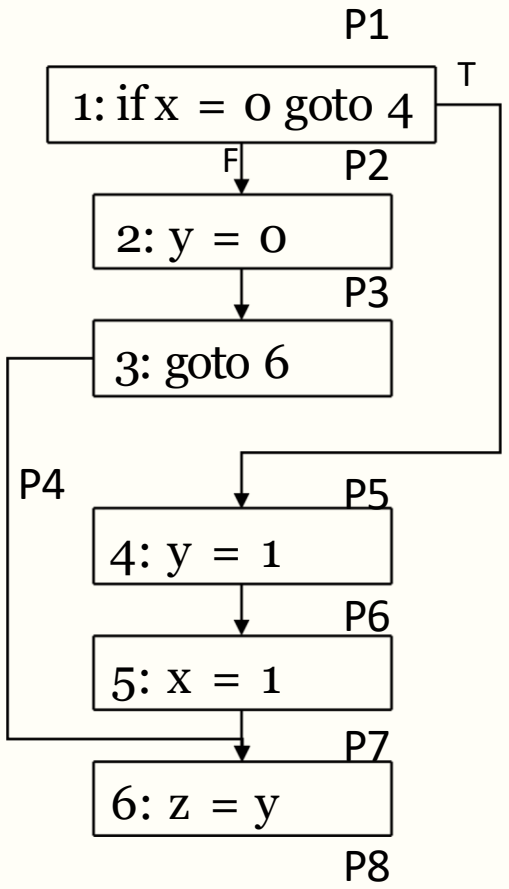
- Exercise: flow function for a conditional branch testing whether x < 0

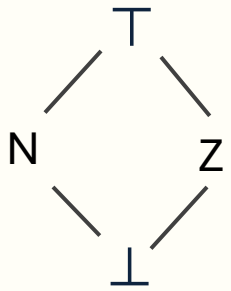
Zero Analysis: Flow Sensitively

❑Example: Branching Code

$$
\begin{aligned}
1:&\quad \text{if } x = 0 \text{ goto } 4\\
2:&\quad y := 0\\
3:&\quad \text{goto } 6\\
4:&\quad y := 1\\
5:&\quad x := 1\\
6:&\quad z := y
\end{aligned}
$$







	x	y	z
P1	⊥	⊥	⊥
P2	N	⊥	⊥
P3	N	Z	⊥
P4	N	Z	⊥
P5	Z	⊥	⊥
P6	Z	N	⊥
P7	N	N	⊥
P8			

States at P4 and P7 meet before line 6

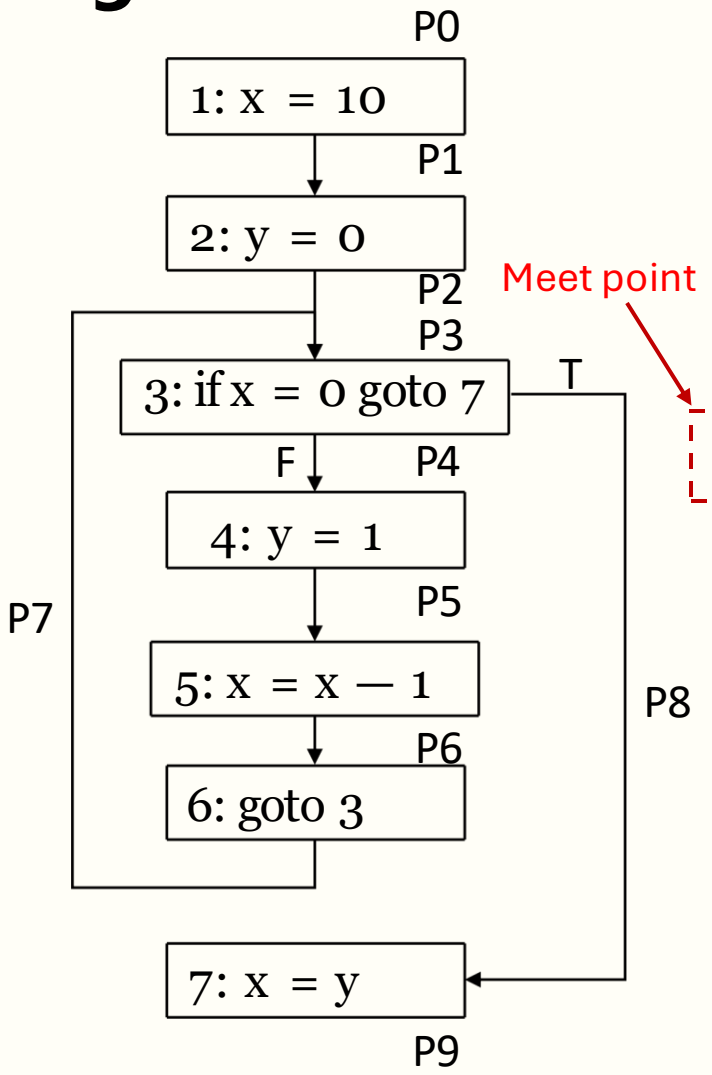
$\sqcup(\sigma_{P4}, \sigma_{P7}) = \{x \mapsto N, y \mapsto \top, z \mapsto \bot\}$

	x	y	z
P1	⊥	⊥	⊥
P2	N	⊥	⊥
P3	N	Z	⊥
P4	N	Z	⊥
P5	Z	⊥	⊥
P6	Z	N	⊥
P7	N	N	⊥
P8	N	⊤	⊤

Zero Analysis: Flow Sensitively

❑Example: Looping Code

$$
\begin{aligned}
1: &\quad x := 10 \\
2: &\quad y := 0 \\
3: &\quad \text{if } x = 0 \text{ goto } 7 \\
4: &\quad y := 1 \\
5: &\quad x := x - 1 \\
6: &\quad \text{goto } 3 \\
7: &\quad x := y
\end{aligned}
$$



Meet point

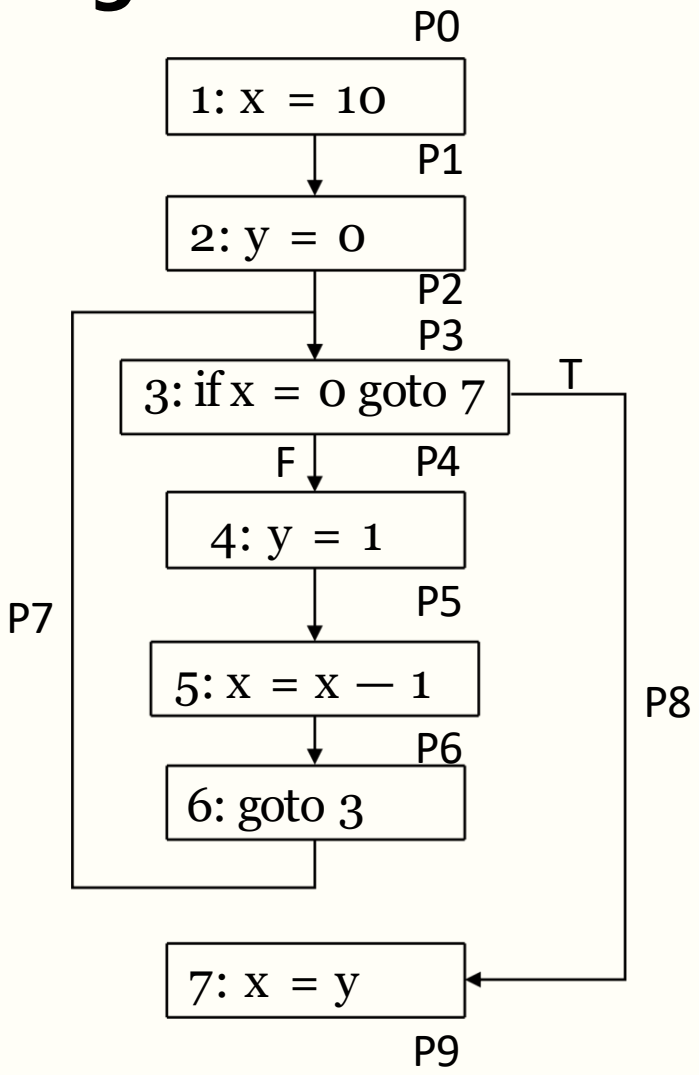
Reach a fixed point!

	Init		1st Time		2nd Time		3rd Time	
	x	y	x	y	x	y	x	y
P0	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
P1	⊥	⊥	N	⊥	N	⊥	N	⊥
P2	⊥	⊥	N	Z	N	Z	N	Z
P3	⊥	⊥	N	Z	T	T	T	T
P4	⊥	⊥	N	Z	T	T	T	T
P5	⊥	⊥	N	N	T	N	T	N
P6	⊥	⊥	T	N	T	N	T	N
P7	⊥	⊥	T	N	T	N	T	N
P8	⊥	⊥	Z	Z	Z	T	Z	T
P9	⊥	⊥	Z	Z	T	T	T	T

Zero Analysis: Flow Sensitively

❑Example: Looping Code

$$
\begin{aligned}
1: &\quad x := 10 \\
2: &\quad y := 0 \\
3: &\quad \text{if } x = 0 \text{ goto } 7 \\
4: &\quad y := 1 \\
5: &\quad x := x - 1 \\
6: &\quad \text{goto } 3 \\
7: &\quad x := y
\end{aligned}
$$



P0: $\sigma_0^{i+1} = \sigma_0^i$

P0 → P1: $\sigma_1^{i+1} = f_z[\![\mathrm{x} := 10]\!](\sigma_0^i)$

P1 → P2: $\sigma_2^{i+1} = f_z[\![\mathrm{y} := 0]\!](\sigma_1^i)$

P3: $\sigma_3^{i+1} = \sqcup \{\sigma_2^i, \sigma_7^i\}$

P3 → P4: $\sigma_4^{i+1} = f_z[\![\text{if x = 0 goto 7}]\!]_F(\sigma_3^i)$

P4 → P5: $\sigma_5^{i+1} = f_z[\![\mathrm{y} := 1]\!](\sigma_4^i)$

P5 → P6: $\sigma_6^{i+1} = f_z[\![\mathrm{x} := \mathrm{x} + 1]\!](\sigma_5^i)$

P6 → P7: $\sigma_7^{i+1} = f_z[\![\text{goto 3}]\!](\sigma_6^i)$

P3 → P8: $\sigma_8^{i+1} = f_z[\![\text{if x = 0 goto 7}]\!]_T(\sigma_3^i)$

P8 → P9: $\sigma_9^{i+1} = f_z[\![\mathrm{x} := \mathrm{y}]\!](\sigma_8^i)$

$$(\sigma_0^{i+1}, \sigma_1^{i+1}, \cdots, \sigma_n^{i+1}) = f_z(\sigma_0^i, \sigma_1^i, \cdots, \sigma_n^i)$$

Zero Analysis: Flow Sensitively

❑Example: Looping Code

- Let $\Sigma^i = (\sigma_0^i, \sigma_1^i, \cdots, \sigma_n^i)$, then $(\sigma_0^{i+1}, \sigma_1^{i+1}, \cdots, \sigma_n^{i+1}) = f_z(\sigma_0^i, \sigma_1^i, \cdots, \sigma_n^i)$ is equivalent to $\Sigma^{i+1} = f_z(\Sigma^i)$
- Let $\Sigma^0 = (\bot, \bot, \cdots, \bot)$, we compute

$$\Sigma^1 = f_z(\Sigma^0) \quad \Sigma^2 = f_z(\Sigma^1)$$

$$\Sigma^3 = f_z(\Sigma^2) \quad \Sigma^2 = \Sigma^3$$

- We stop when finding a fixed point:

Correctness theorem:
If data-flow analysis is well designed, then any fixed point of the analysis is **sound**.

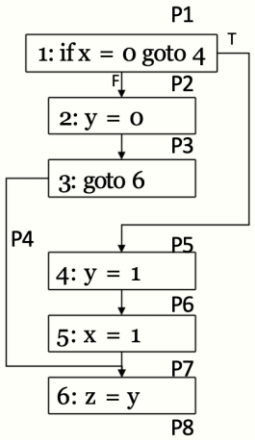
	Init		1st Time		2nd Time		3rd Time	
	x	y	x	y	x	y	x	y
P0	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
P1	⊥	⊥	N	⊥	N	⊥	N	⊥
P2	⊥	⊥	N	Z	N	Z	N	Z
P3	⊥	⊥	N	Z	T	T	T	T
P4	⊥	⊥	N	Z	T	T	T	T
P5	⊥	⊥	N	N	T	N	T	N
P6	⊥	⊥	T	N	T	N	T	N
P7	⊥	⊥	T	N	T	N	T	N
P8	⊥	⊥	Z	Z	Z	T	Z	T
P9	⊥	⊥	Z	Z	T	T	T	T

Σ^0 $\Sigma^1 = f_z(\Sigma^0)$ $\Sigma^2 = f_z(\Sigma^1)$ $\Sigma^3 = f_z(\Sigma^2)$

数据流分析（Data Flow Analysis）

2020年图灵奖获得者

❑Monotone framework（By John B. Kam, Jeffrey D. Ullman, 1977）



- ➢A **CFG** to be analyzed
  - each node v_i representing an instruction $inst_i$
  - Each edge e_i related to two nodes, representing a **program point** P_i
  - In case a node v_j has multiple incoming edges, we introduce an additional program point P_j' for ease of meet operation
- ➢An abstraction function α
  - maping concrete domain values to abstract domain values
  - abstract domain forms a **finite-height complete lattice** L
- ➢A **variable σ_i** associated with each program point P_i
  - Value of σ_i, $[\![\sigma_i]\!] \in L$
- ➢A **monotone function f** $\in L^n \to L^n$:
  - $\sigma_i = f_i(\sigma_1, \sigma_2, \cdots, \sigma_n)$; $(\sigma_1, \sigma_2, \cdots, \sigma_n) = f(\sigma_1, \sigma_2, \cdots, \sigma_n)$
- ➢And **initial value** for each variable σ_i, usually the $\perp$

数据流分析（Data Flow Analysis）

❑Monotone framework

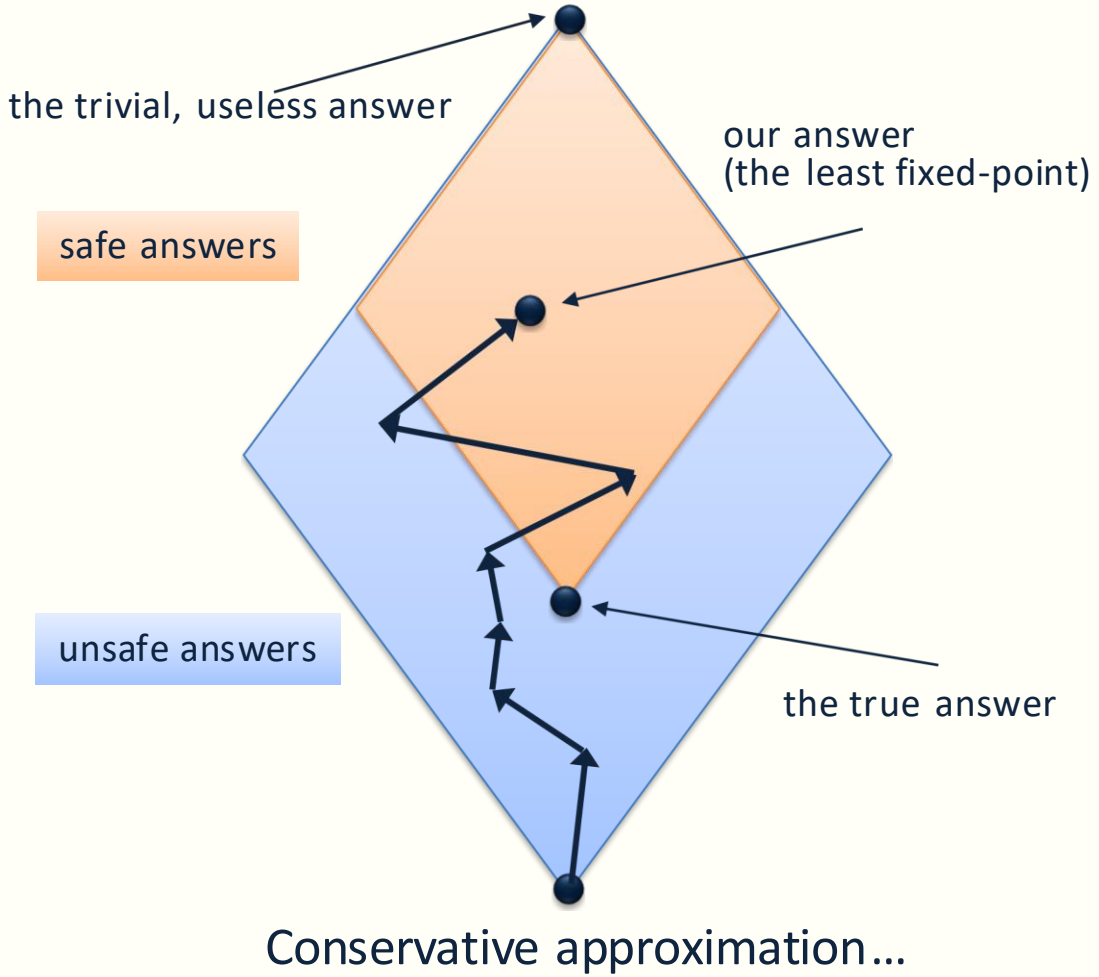
- ➢Compute the data flow analysis via the fixed-point algorithm
  - L is a complete lattice with finite height, so is the product lattice L^n
  - Flow function f_i: $\sigma_i = f_i(\sigma_1, \sigma_2, \cdots, \sigma_n)$ is monotone, so is f, where; $f(\sigma_1, \cdots, \sigma_n) = (\sigma_1, \cdots, \sigma_n) = (f_1(\sigma_1, \cdots, \sigma_n), f_2(\sigma_1, \cdots, \sigma_n), \cdots, f_3(\sigma_1, \cdots, \sigma_n))$
  - Successive applications of f results in an ascending chain until reaching a fixed-point

- ➢This solution gives an answer from L for each program point
  - A solution is always a fixed point of $f: L^n \to L^n$
  - The obtained solution is the least fixed point of $f: L^n \to L^n$
  - There is a unique least fixed point, which is the most precise

The Naïve Algorithm

```
x = (⊥, ⊥, …, ⊥);
do {
    t = x;
    x = f(x);
} while (x≠t);
```

	$f^0(\bot, \bot, \ldots, \bot)$	$f^1(\bot, \bot, \ldots, \bot)$	…	$f^k(\bot, \bot, \ldots, \bot)$
1	⊥	$f_1(\bot, \bot, \ldots, \bot)$	…	…
2	⊥	$f_2(\bot, \bot, \ldots, \bot)$	…	…
…	…	…	…	…
n	⊥	$f_n(\bot, \bot, \ldots, \bot)$	…	…



- **Computing each new entry using the previous column**
  - Without using the entries that have already been computed in the current column!
  - And many entries are likely unchanged from one column to the next!

Less efficient!

Chaotic iteration

- **Compute** $f(\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_n}) = (f_1(\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_n}), \ldots, f_n(\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_n}))$

$\sigma_1 = \bot$; … $\sigma_n = \bot$;
while $((\sigma_1, \ldots, \sigma_n) \neq f(\sigma_1, \ldots, \sigma_2))$ {
  pick i nondeterministically such that $\sigma_i \neq f_i(\sigma_1, \ldots, \sigma_n)$
  $\sigma_i = f_i(\sigma_1, \ldots, \sigma_n)$;
}

→ How to pick effectively?

- Require a higher number of iterations but less work in each iteration
- In the i^{th} iteration, we have $\sigma_j^{i-1} \sqsubseteq \sigma_j^i$ for each $j \in [1, n]$
- Chaotic iteration eventually terminates at the least fixed point by Kleene's theorem

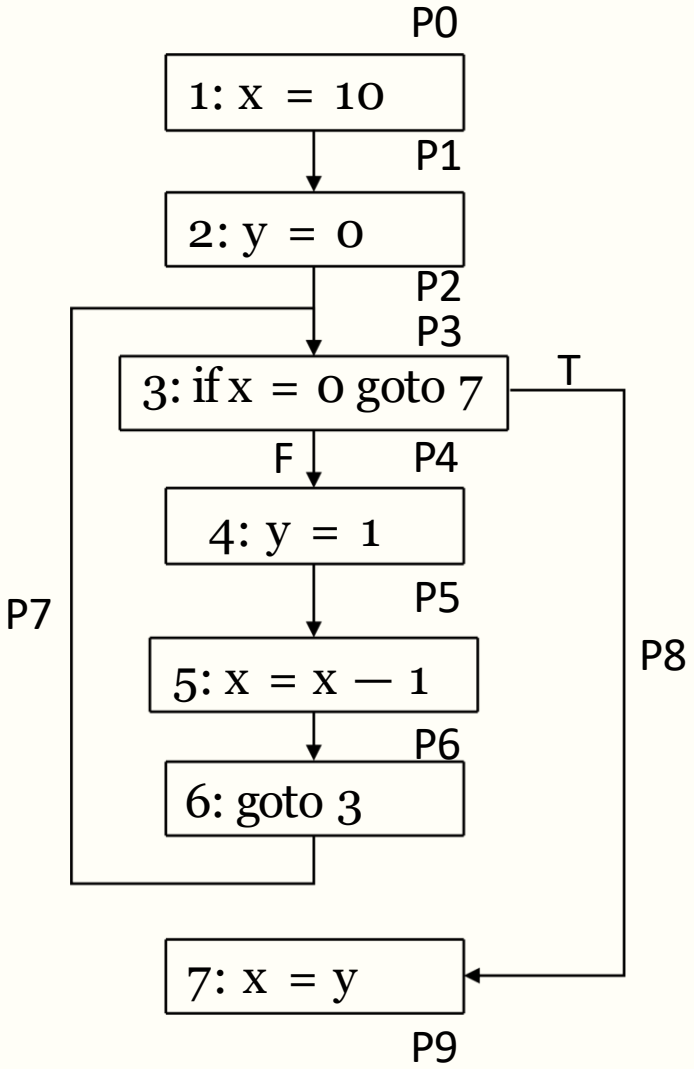
Kildall's Algorithm

- **The policy of order to process worklist**
  - Random?
  - Queue?
  - Stack?
  - All are valid!
  - Some are better in practice:
    - Topological sorts are nice
    - Explore loops inside out
    - Reverse postorder!

```
worklist = ∅
foreach program point P_i in CFG:
  ⟦σ_i⟧ = ⊥
⟦σ_0⟧ = initialDataflowInformation
add σ_0 to worklist

while worklist ≠ ∅:
  take one σ_i off the worklist
  tmp = ⟦σ_i⟧
  ⟦σ_i⟧ = f_i(σ_1, σ_2, ⋯, σ_n)
  if tmp ≠ ⟦σ_i⟧:
    foreach program point P_j ∈ succ(P_i)in CFG:
      add σ_j to worklist if it is not in it
```

Exercise: Apply Kildall's Algorithm



Suppose the worklist is a stack:

$\sigma_0 = \{ x \mapsto \bot, y \mapsto \bot \}$
$\sigma_1 = \{x \mapsto N, y \mapsto \bot\}$
$\sigma_2 = \{x \mapsto N, y \mapsto Z\}$
$\sigma_3 = \{x \mapsto N, y \mapsto Z\}$
$\sigma_4 = \{x \mapsto N, y \mapsto Z\}$
$\sigma_5 = \{x \mapsto N, y \mapsto N\}$
$\sigma_6 = \{x \mapsto \top, y \mapsto N\}$
$\sigma_7 = \{x \mapsto \top, y \mapsto N\}$
$\sigma_3 = \{x \mapsto \top, y \mapsto \top\}$
$\sigma_4 = \{x \mapsto \top, y \mapsto \top\}$
$\sigma_5 = \{x \mapsto \top, y \mapsto N\}$
$\sigma_6 = \{x \mapsto \top, y \mapsto N\}$
$\sigma_7 = \{x \mapsto \top, y \mapsto N\}$
$\sigma_3 = \{x \mapsto \top, y \mapsto \top\}$
$\sigma_8 = \{x \mapsto Z, y \mapsto \top\}$
$\sigma_9 = \{x \mapsto \top, y \mapsto \top\}$

The number of flow function computations is significantly reduced.

Termination of Kildall's Algorithm

- Terminates at the least fixed point by Kleene's theorem
- Assume the designed flow functions are monotonic

Monotonicity A function f is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

- 终止性证明:
  - $f_i(\sigma_1, \sigma_2, \cdots, \sigma_n) \sqsubseteq [\![\sigma_i]\!]$ for all i
  - if $f_i(\sigma_1, \sigma_2, \cdots, \sigma_n) \neq [\![\sigma_i]\!]$, P_i的后续程序点将被放入worklist中
  - 假设Kildall算法不能终止，则不断会有某个程序点P_j，$f_j(\sigma_1, \sigma_2, \cdots, \sigma_n) \sqsubset [\![\sigma_j]\!]$, 则$L^n$构成的product lattice高度无限，与假设L为有限高度格矛盾

```
worklist = ∅
foreach program point P_i in CFG:
  [[σ_i]] = ⊥
[[σ_0]] = initialDataflowInformation
add σ_0 to worklist

while worklist ≠ ∅:
  take one σ_i off the worklist
  tmp = [[σ_i]]
  [[σ_i]] = f_i(σ_1, σ_2, ⋯, σ_n)
  if tmp ≠ [[σ_i]]:
    foreach program point P_j ∈ succ(P_i)in CFG:
      add σ_j to worklist if it is not in it
```

Complexity of Kildall's Algorithm

- Let h be the lattice height of L
- CFG has n program points, e edges
- Complexity: $O(\max(e,n) * h)$

```
worklist = ∅
foreach program point P_i in CFG:
  ⟦σ_i⟧ = ⊥
⟦σ_0⟧ = initialDataflowInformation
add σ_0 to worklist

while worklist ≠ ∅:
  take one σ_i off the worklist
  tmp = ⟦σ_i⟧
  ⟦σ_i⟧ = f_i(σ_1, σ_2, ⋯, σ_n)
  if tmp ≠ ⟦σ_i⟧:
    foreach program point P_j ∈ succ(P_i) in CFG:
      add σ_j to worklist if it is not in it
```

$O(n)$

$O(1)$

$O(n * h)$

$O(e * h)$ in total (across the while loop)

- 格L^n的高度是$n * h$
- 所以算法最后两行最多执行$n * h$次
- 对整个while循环来说，倒数第二行共遍历$O(e * h)$个程序点
- 故总复杂度应为 $O(\max(e,n) * h)$

Program Trace

❑Configuration $c = \langle E, P_i \rangle$:

➢ E is the environment: $Var \mapsto Int$; and P_i is the program point

❑Program Trace

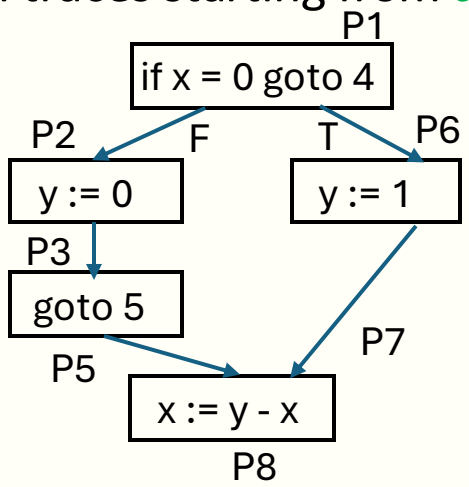
A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, \cdots\}$ of program configurations, where $c_0 = \langle E_0, P_1 \rangle$ is called the initial configuration, and for every $i \geq 0$ we have $P \vdash c_i \rightsquigarrow c_{i+1}$.

❑Example

Leave as an exercise

Write two program traces starting from $c_0 = \langle x \to 0, y \to 1, P_1 \rangle$ *and* $c_0 = \langle x \to 1, y \to 0, P_1 \rangle$

```
1: if x = 0 goto 4
2: y := 0
3: goto 5
4: y := 1
5: x := y - x
```



c_0=$\langle x \to 0, y \to 1, P_1 \rangle$, $\langle x \to 0, y \to 1, P_6 \rangle$,
$\langle x \to 0, y \to 1, P_7 \rangle$, $\langle x \to 1, y \to 1, P_8 \rangle$

Precision

- **Optimal Precision (OPT):**
  - $\Gamma(\mathrm{j}) = \{\mathrm{c} \mid \mathrm{c} = \langle \mathrm{E}, \mathrm{j} \rangle \wedge \mathrm{c} \in \mathrm{Traces(P)}\}$ collects all possible concrete configurations observed at program point j
  - Optimal precision: $\langle \sigma_n \mid \mathrm{n} \in \mathrm{P} \rangle$, where $\sigma_i = \sqcup_{c \in \Gamma(i)} \alpha(c)$
- **Merge Over Paths (MOP):**
  - $\mathrm{MOP(i)} = \sqcup\{\sigma \mid \langle \sigma, \mathrm{i} \rangle \in \mathrm{Some}\ \Pi \ \mathrm{for}\ P\}$, where i is a program point
  - $\Pi = \langle \sigma_1, n_1 \rangle, \langle \sigma_2, n_2 \rangle, \cdots$ is a sequence, where program point n_i forms a path $\pi = n_1, n_2, \cdots$ in CFG and $\sigma_i = \mathrm{f}[\![n_{i-1} \to n_i]\!](\sigma_{i-1})$
- **Least Fixed Point (LFP):**
  - $\mathcal{F}(\Sigma^*) = \Sigma^*, \forall\, \Sigma\, (\mathcal{F}(\Sigma) = \Sigma) \Longrightarrow (\Sigma\hat{}\, * \sqsubseteq \Sigma)$
- **Relative precision:**

$$\mathrm{OPT} \sqsubseteq \mathrm{MOP} \sqsubseteq \mathrm{LFP} \sqsubseteq \mathrm{FP} \sqsubseteq \top$$

Soundness

The result $\langle \sigma_i \mid P_i \in P \rangle$ of a program analysis running on program P is ***sound*** iff, for all traces $T = \{c_0, c_1, \cdots\}$ of P, for all i such that $0 \leq i < \text{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_i$.

❑Local Soundness:

A flow function f is ***locally sound*** iff $P \vdash c_i = (E_i, P_i) \rightsquigarrow (E_{i+1}, P_{i+1})$ and $\alpha(c_i) \sqsubseteq \sigma_i$ and $f[\![P_i \rightarrow P_{i+1}]\!](\sigma_i) = \sigma_{i+1}$ implies $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$.

❑Global Soundness

Theorem 2 (A fixed point of a locally sound analysis is globally sound). *If a dataflow analysis's flow function f is monotonic and locally sound, and for all traces T we have $\alpha(c_0) \sqsubseteq \sigma_0$ where σ_0 is the initial analysis information, then any fixed point $\{\sigma_n \mid n \in P\}$ of the analysis is sound.*

Proof. To show that the analysis is sound, we must prove that for all program traces, every program configuration in that trace is correctly approximated by the analysis results. We consider an arbitrary program trace T and do the proof by induction on the program configurations $\{c_i\}$ in the trace.

How to design a data flow analysis

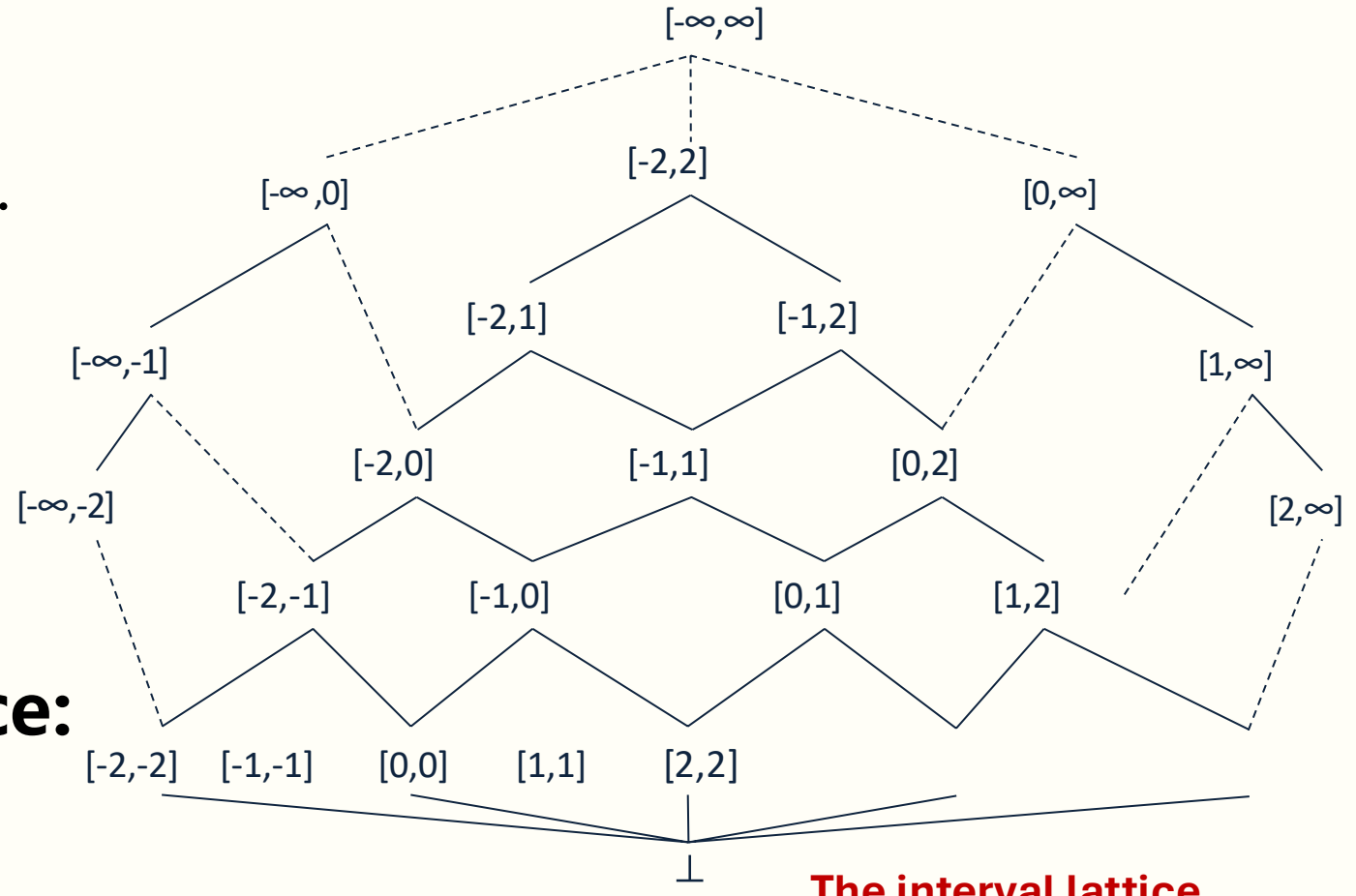
- **Design the abstraction α:**
  - concrete domain, abstract domain and the lattice
- **Design the flow functions $\mathcal{F} = (f_1, f_2, \cdots, f_n)$**
  - Make sure each f_i is monotonic
  - Make sure the local soundness of each f_i
- **Design the meet operation $\sqcup$**
- **Apply the monotone framework to compute the solution, which should be the least fixed point**

DFA Application: Interval analysis

- **Compute upper and lower bounds for integers**
- **Possible applications:**
  - array bounds checking,
  - integer representation, …
- **Interval Lattice:**
  - Ordered by inclusion:

$[l_1, r_1] \sqsubseteq [l_2, r_2]$ iff $l_2 \leq l_1 \wedge r_1 \leq r_2$

- **Interval Analysis Lattice:**
  - $Map\ lattice$: $Var \rightarrow L$
  - We use $lift(Var \rightarrow L)$



The interval lattice

DFA Application: Interval analysis

❑Meet Operation ⊔

- ⊔ I = UI, e.g., ⊔ $\{[-20,-10],[-15,-9],[3,7],[4,18]\} = \{[-20,-9],[3,18]\}$

❑Flow Functions $\mathcal{F}$:

$\boldsymbol{Inst}$ ::= $x \coloneqq n \mid x \coloneqq y \mid x \coloneqq y\ op\ z \mid \text{goto } n \mid \text{if } x\ \text{op}_\text{r}\ 0 \text{ goto } n$

$f_\text{I}[\![x \coloneqq n]\!](\sigma) = \sigma[x \mapsto [n,n]]$

$f_\text{I}[\![x \coloneqq y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$

$f_\text{I}[\![\text{goto } n]\!](\sigma) = \sigma$

$f_\text{I}[\![\text{if } op_r\ 0 \text{ goto } n]\!](\sigma) = \sigma$

$f_\text{I}[\![x \coloneqq y\ op\ z]\!](\sigma) = \overline{\text{op}}(\sigma(\text{y}), \sigma(\text{z}))$

Abstract Operator: $\overline{\text{op}}([l_1,r_1],[l_2,r_2]) = [\min_{x\in[l_1,r_1],y\in[l_2,r_2]} x\ op\ y,\ \max_{\text{x}\in[l_1,r_1],\text{y}\in[l_2,r_2]} \text{x op y}]$

❑Compute the fixed-point : $(\boldsymbol{\sigma_1},\cdots,\boldsymbol{\sigma_n}) = \boldsymbol{\mathcal{F}}(\boldsymbol{\sigma_1},\cdots,\boldsymbol{\sigma_n})$

- $\sigma_i = f_\text{I}^i(\boldsymbol{\sigma_1},\cdots,\boldsymbol{\sigma_n})$; compute $f_I^i(\bot)$ for $i = 0,1,\cdots$
- **Does the least fixed point exist ?**
- The lattice L has *infinite* height: $[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \cdots$

Does the least fixed point exist?

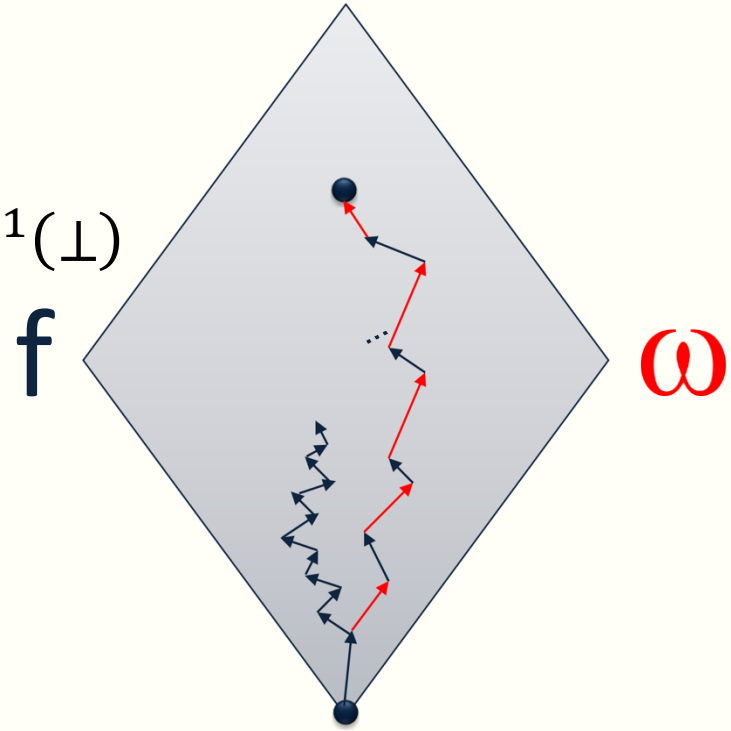
- **The lattice L has infinite height:**
  - Kleene's fixed-point theorem does not apply here
- **Tarski's fixed-point theorem:**

> Let $(L, \sqsubseteq)$ be a complete lattice and $f: L \to L$ be monotone increasing. Then the set of all fixed points of f is a complete lattice with respect to $\sqsubseteq$ (Tarski 1995).

  - Corollary: f has a unique greatest fixed point and a unique least fixed point.
  - Proof: https://pauldelatte.github.io/files/L3.pdf

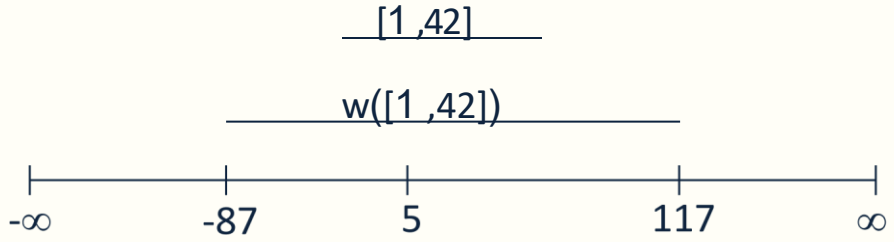
Widening: solution to find the fixed point

- **Widening could be very useful in interval analysis.**
  - 通过降低结果的精度来加快收敛速度
- **Define a widening function $\omega: L \to L$**
  - ω is an extensive and monotone function (so is $\omega \circ f$),
  - the sub-lattice $\omega(L)$ has finite height
- **$(\omega \circ f)^i(\bot)$ for $i = 0, 1, \ldots$ converges**
  - Let $f_\omega = (\omega \circ f)^k(\bot)$ where $(\omega \circ f)^k(\bot) = (\omega \circ f)^{k+1}(\bot)$
  - $\omega \circ f$ becomes the new flow function
- **lfp(f) $\sqsubseteq f_\omega$ holds**
  - i.e., f_ω is a safe approximation of $lfp(f)$
  - follows from Tarski's fixed-point theorem



Simple Widening for Interval Analysis

- **Widening function $\omega: L \to L$:**
  - Given a fixed finite set S: 降低格的高度
    - must contain $-\infty, +\infty$ (to retain the ⊤),
    - typically seeded with all integer constants in the given program
  - **Idea**: **find the nearest enclosing allowed interval**
  - **ω** is defined pointwise on $L = (Var \to Interval)^n$
    - $\omega([a,b]) = [\max\{i \in S \mid i \leq a\}, \min\{i \in S \mid b \leq i\}]$
    - $\omega(\bot) = \bot$



Simple Widening for Interval Analysis

❑Example

$f_{\mathrm{I}}[\![x := n]\!](\sigma) = \sigma[x \mapsto [n,n]]$

$f_{\mathrm{I}}[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$

$f_{\mathrm{I}}[\![\text{goto } n]\!](\sigma) = \sigma$

$f_{\mathrm{I}}[\![\text{if } x\ op_r\ 0 \text{ goto } n]\!](\sigma) = \sigma$

$f_{\mathrm{I}}[\![x := y\ op\ z]\!](\sigma) = \overline{\mathrm{op}}(\sigma(\mathrm{y}), \sigma(\mathrm{z}))$

Abstract Operator: $\overline{\mathrm{op}}([l_1,r_1],[l_2,r_2]) = [\min_{x\in[l_1,r_1],y\in[l_2,r_2]} x\ op\ y,\ \max_{x\in[l_1,r_1],y\in[l_2,r_2]} x\ \mathrm{op}\ y]$

❑States at the while

```
1 y = 0;
2 x = 7;
3 x = x + 1;
4 while (input) {
5   x = 7;
6   x = x + 1;
7   y = y + 1;
8 }
```

不使用Widening算子：

```
[x → [8,8],y → [0,0]]
[x → [8,8], y → [0,1]]
[x → [8,8], y → [0,2]]
[x → [8,8], y → [0,3]]
…
```

不收敛！

使用Widening算子：

S = {-∞, 0, 1, 7, ∞}

```
[x → [7, ∞],y → [0,0]]
[x → [7,∞ ], y → [0,1]]
[x → [7,∞ ], y → [0,7]]
[x → [7,∞ ], y → [0,∞ ]]
```

收敛快，但不精确

More powerful widening

- **Simple widening happens at every interval and node**
- **No need to widen intervals that are "stable"**
  - E.g., no need to widen if there are no "cycles" in the dataflow
- **More powerful widening ∇: $L \times L \to L$**
  - extensive in both arguments, i.e., $x_1 \sqsubseteq x_1 \nabla x_2 \ \wedge \ x_2 \sqsubseteq x_1 \nabla x_2$
  - for all increasing chains $z_0 \sqsubseteq z_1 \sqsubseteq \cdots$, the sequence $y_0 = z_0, \cdots, y_{i+1} = y_i \nabla z_{i+1}, \cdots$ converges
- **New fixed-point solver by computing**
  - $t_0 = (\bot, \cdots, \bot)$, $t_{i+1} = t_i \nabla f(t_i)$ until convergence
- **Theorem: $lfp(f) \ \sqsubseteq \ x_k$ for some k such that $x_{k+1} = x_k$**

More powerful widening: Example

S = {-∞, 0, 1, 7, ∞}

❑Interval Analysis

- ➢Given the fixed finite set S, define ∇: pointwise down to intervals
  - $\perp \nabla y = y$, $x \nabla \perp = x$, $[a_1, b_1] \nabla [a_2, b_2] = [a, b]$
  - $a = a_1$ if $a_1 \leq a_2$ else $\max\{i \in S \mid i \leq a_2\}$, and $b = b_1$ if $b_2 \leq b_1$ else $\min\{i \in S \mid b_2 \leq i\}$

```
{x →⊥, y →⊥}
1 y = 0;
{x →⊥, y → [0,0]}
2 x = 7;
{x → [7,7], y → [0,0]}
3 x = x + 1;
{x → [7,∞], y → [0,0]}
4 while (input) {
{x → [7,∞], y → [0,∞]}
5     x = 7;
{x → [7,7], y → [0,∞]}
6     x = x + 1;
{x → [7,∞], y → [0,∞]}
7     y = y + 1;
{x → [7,∞], y → [0,∞]}
8 }
```

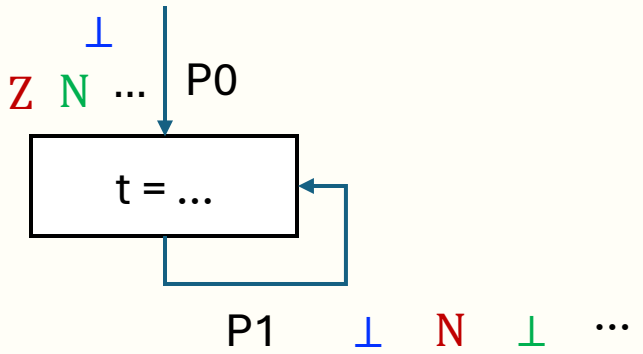
收敛更快，也精度!

Discuss why ∇ should be extensive

- **$\nabla: L \times L \to L$ is extensive**
  - i.e., $\forall\, x_1, x_2 : x_1 \sqsubseteq x_1 \nabla x_2 \;\wedge\; x_2 \sqsubseteq x_1 \nabla x_2$
- $x_2 \sqsubseteq x_1 \nabla x_2$:
  - ensures the termination of fixed-point computation
- $x_1 \sqsubseteq x_1 \nabla x_2$:
  - The value in the next round could be smaller than the previous round, potentially leading to an incessant oscillation (震荡不终止)
  - Example:

$$x \nabla y = \begin{cases} Z & y = \bot \\ y & \text{otherwise} \end{cases}$$

$$f[\![v]\!](\sigma) = \begin{cases} N & \sigma(t) = Z \\ \bot & \text{otherwise} \end{cases}$$



Further Improvement

- **Why widening at every interval and node**
- **No need to widen if alreay "stable"**
  - divergence can only appear in presence of recursive dataflow constraint
  - No need to widen if no "cycles" in the dataflow
  - Sufficient to "break the cycles", perform widening only at, loop heads in the CFG

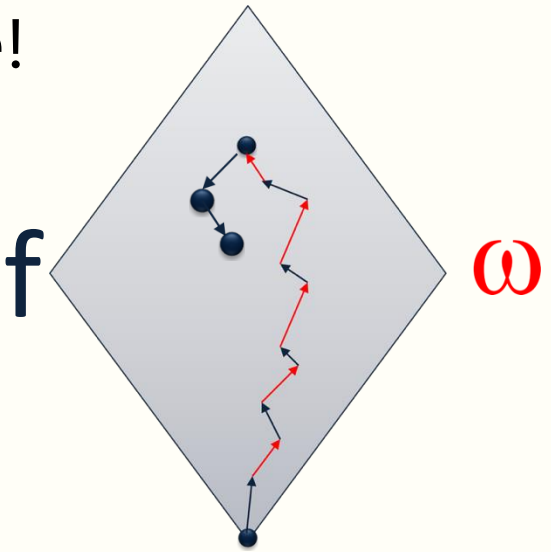
Widening only at loop head

$S = \{-\infty, 0, 1, 7, \infty\}$

```
   {x → ⊥, y → ⊥}
1  y = 0;
   {x → ⊥, y → [0,0]}
2  x = 7;
   {x → [7,7], y → [0,0]}
3  x = x + 1;
   {x → [8,8], y → [0,0]}
4  while (input) {
∇  {x → [7,∞], y → [0,∞]}
5    x = 7;
   {x → [7,7], y → [0,∞]}
6    x = x + 1;
   {x → [8,8], y → [0,∞]}
7    y = y + 1;
   {x → [8,8], y → [1,∞]}
8  }
```

Narrowing：improve the precision

- **Widening generally shoots over the target**
  - lfp(f) $\sqsubseteq f_\omega = (\omega \circ f)^k(\bot)$
- ***Narrowing* may improve the result by applying f**
  - $f(f_\omega) \sqsubseteq f_\omega$: so applying f again may improve the result!
    - $\mathrm{lfp}(f) = f(\mathrm{lfp}(f)) \sqsubseteq f(f_\omega) \sqsubseteq \omega(\mathrm{f}(f_\omega)) = (\omega \circ \mathrm{f})(f_\omega) = f_\omega$
  - And we also have $lfp(f) \sqsubseteq f(f_\omega)$ so it remains safe!
  - This can be iterated arbitrarily many times
    - $\mathrm{lfp}(f) \sqsubseteq f^i(f_\omega) \sqsubseteq f^{i-1}(f_\omega)$ implies $\mathrm{lfp}(f) \sqsubseteq f^{i+1}(f_\omega) \sqsubseteq f^i(f_\omega)$
      - f is monotonic
    - $\mathrm{lfp}(\mathrm{f}) \sqsubseteq \cdots \sqsubseteq f^i(f_\omega)) \sqsubseteq \cdots \sqsubseteq \mathrm{f}(f_\omega)) \sqsubseteq f_\omega$
  - may diverge, but safe to stop anytime
    - Either prove divergence for a specific application
    - Or just apply f a fixed time



Narrowing不保证收敛，也不保证终止.

Narrowing

$S = \{-\infty, 0, 1, 7, \infty\}$

```
   {x → ⊥, y → ⊥}
1  y = 0;
   {x → ⊥, y → [0,0]}
2  x = 7;
   {x → [7,7], y → [0,0]}
3  x = x + 1;
   {x → [7,∞], y → [0,0]}
4  while (input) {
   {x → [7,∞], y → [0,∞]}
5    x = 7;
   {x → [7,7], y → [0,∞]}
6    x = x + 1;
   {x → [7,∞], y → [0,∞]}
7    y = y + 1;
   {x → [7,∞], y → [1,∞]}
8  }
```

Use widening

```
   {x → ⊥, y → ⊥}
1  y = 0;
   {x → ⊥, y → [0,0]}
2  x = 7;
   {x → [7,7], y → [0,0]}
3  x = x + 1;
   {x → [8,8], y → [0,0]}
4  while (input) {
   {x → [8,8], y → [0,∞]}
5    x = 7;
   {x → [7,7], y → [0,∞]}
6    x = x + 1;
   {x → [8,8], y → [0,∞]}
7    y = y + 1;
   {x → [8,8], y → [1,∞]}
8  }
```

Use narrowing

The result becomes more precise!

Extensions

- **Intrer-procedural data flow analysis**
  - Working on inter-procedural control flow graph
  - Eliminate infeasible paths by using context sensitivity
    - Call-strings, and many others
    - Expensive to compute, poor scalability
- **Path-sensitive data flow analysis**
  - Configuration is further extended with path conditions
  - $\sigma = \langle E, P_i, C \rangle$, where $\mathrm{E}: \mathrm{Var} \rightarrow \mathrm{Abstract\ Values}$, P_i is a program point, and C is the path condition under which P_i could be reachable

Flow insensitive analysis : $\sigma = \mathrm{E}$ Flow sensitive analysis : $\sigma = \langle E, P_i \rangle$

Path sensitive analysis : $\sigma = \langle E, P_i, \mathrm{C} \rangle$ Context sensitive analysis : $E: Var \times Ctxt \rightarrow Abstract\ Values$

Classic Data Flow Analysis

- **We use Zero Analysis as a motivating example of DFA**
- **There are several classic DFA applications:**
  - Integer Sign Analysis
  - Constant Propagation
  - Reaching Definitions
  - Live Variables Analysis
  - Available Expressions
  - Very Busy Expressions
- **We discuss "Very Busy Expressions" here, leaving others for optional assignments**

Very Busy Expressions

- **A (nontrivial) expression is *very busy* if it will definitely be evaluated before its value changes**
  - The expression a*b in the following example is very busy
  - The answer *"very busy"* **must** hold at a given program point
- **Very Busy Expressions are used in the *loop hoisting* pass during compiler optimization**

```
read x;
a = x + 3;
b = x - 1;
while (x > 0) {
  t = a * b;
  print t + x;
  x = x - 1;
}
print a * b;
```

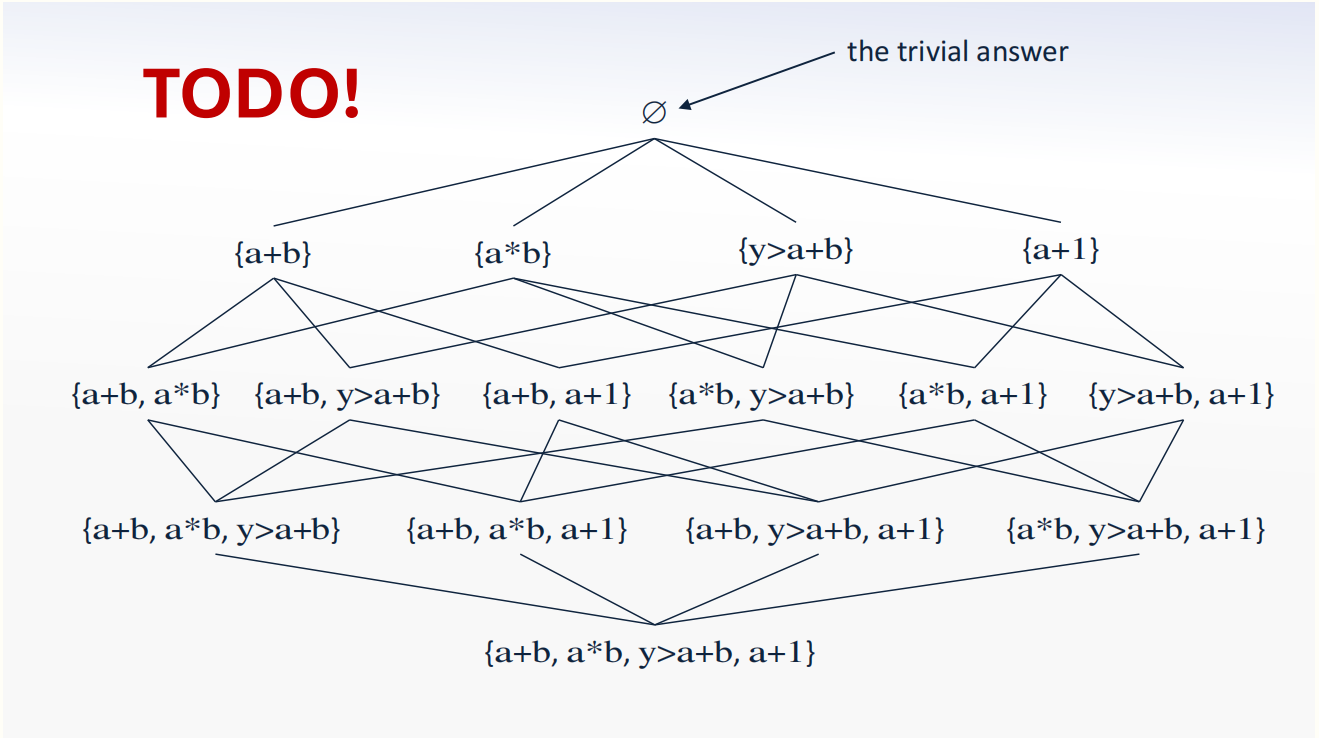
Loop Hoisting

```
read x;
a = x + 3;
b = x - 1;
t = a * b;
while (x > 0) {
  print t + x;
  x = x - 1;
}
print t;
```

A lattice for Very Busy Expressions

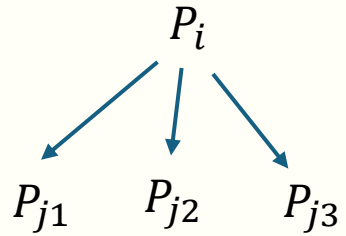
- **A reverse powerset lattice of nontrivial expressions**
  - $L = (\mathcal{P}(x+3, x-1, x>0, a*b, \mathrm{t}+x), \supseteq)$
  - Nontrivial expressions are the values in the abstract domain by α

```
read x;
a = x + 3;
b = x - 1;
while (x > 0) {
  t = a * b;
  print t + x;
  x = x - 1;
}
print a * b;
```



Very Busy Expressions

- **A backward data flow analysis**
  - At the CFG entry, we know nothing about very busy expressions
  - At the CFG exit, we assume all expressions appear are very busy
- **Meet operator ⊔:**
  - An expression is very busy at P_i if it is very busy at all P_i's successors
  - $\sigma_i = \sqcup_{p_j \in succ(p_i)} (\sigma_j) = \cap_{p_j \in succ(p_i)} (\sigma_j)$



- **Flow functions:** ***Inst*** ::= $x := n \mid x := y \mid x := y\ op\ z \mid$ goto $n \mid$ if x op$_r$ 0 goto $n \mid$ read x; $\mid$ print x;
  - $S \downarrow x$: retain expressions without the variable x from the set S

$f_I[\![x := n]\!](\sigma) = \sigma \downarrow x$

$f_I[\![x := y]\!](\sigma) = \sigma \downarrow x$

$f_I[\![x := y\ op\ z]\!](\sigma) = (\sigma \cup \{y\ op\ z\}) \downarrow x$

$f_I[\![\text{if } x\ op_r\ 0 \text{ goto } n]\!](\sigma) = \sigma \cup \{x\ op_r\ 0\}$

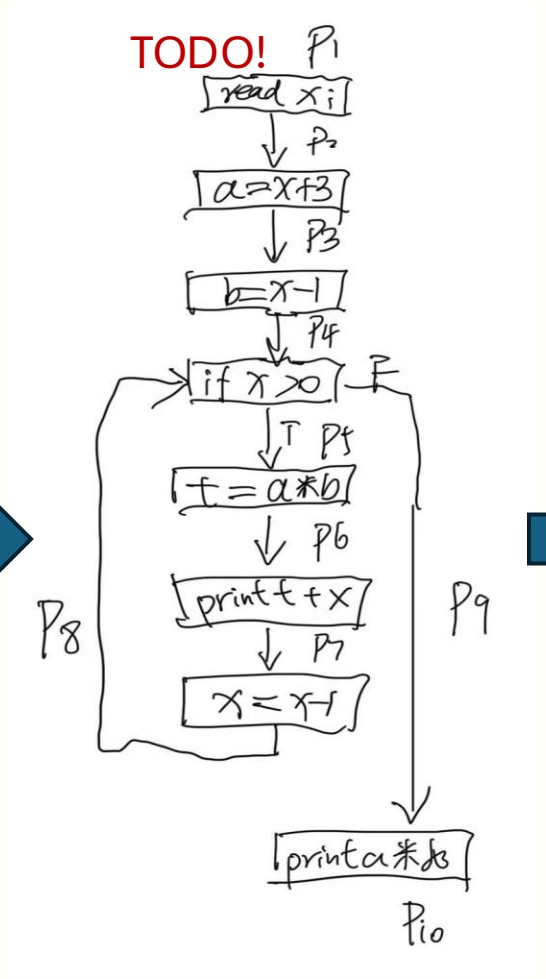
$f_I[\![\text{goto } n]\!](\sigma) = \sigma$

$f_I[\![\text{read x}]\!](\sigma) = \sigma \downarrow x$

$f_I[\![\text{print x}]\!](\sigma) = \sigma$

Very Busy Expressions

```
read x;
a = x + 3;
b = x - 1;
while (x > 0) {
  t = a * b;
  print t + x;
  x = x - 1;
}
print a * b;
```



P10	$\{x+3, x-1, x>0, a*b, t+x\}$
P9	$\{x+3, x-1, x>0, a*b, t+x\}$
P8	$\{x+3, x-1, x>0, a*b, t+x\}$
P7	$\{a*b\}$
P6	$\{a*b, \mathrm{t}+\mathrm{x}\}$
P5	$\{a*b\}$
P8	$\{a*b\}$
P4	$\{a*b\}$
P3	$x-1$
P2	$x-1, x+3$
P1	$\emptyset$

Classifying data flow analyses

- **A forward analysis**
  - Computes information about the past behavior
  - Start from CFG's entry
- **A backward analysis**
  - Computes information about the future behavior
  - Start from CFG' exit
- **A may analysis:**
  - Describes information that is possibly true; an over-approximation
- **A must analysis:**
  - Describes information that is definitely true; An under-approximation
- **Example:**
  - Zero Analysis: a forward, may analysis
  - Very Busy Expressions: a backward, must analysis

(Optional)作业：Classifying DFA

❑查询相关资料，为如下数据流分析应用分类

- ➢Integer Sign Analysis
- ➢Constant Propagation
- ➢Reaching Definitions
- ➢Live Variables Analysis
- ➢Available Expressions

	Forward	Backward
May	Zero analysis	
Must		Very Busy Expressions

❑挑选1-2个应用写出其抽象域构成的格、流函数以及Meet操作

(Optional）作业：熟悉数据流分析框架

- **学习LLVM中的数据流分析框架实现**
  - https://github.com/llvm/llvm-project/
  - mlir/include/mlir/Analysis/DataFlowFramework.h
  - 制作slides分享对框架的理解
- **学习LLVM中数据流分析应用的实现**
  - mlir/include/mlir/Analysis/DataFlow
  - LLVM中提供了一些常见数据流分析应用的实现，请选择一个学习，并制作slides分享学习体会