# Lecture 6 – Widening and Narrowing

## SWS128: Software Analysis

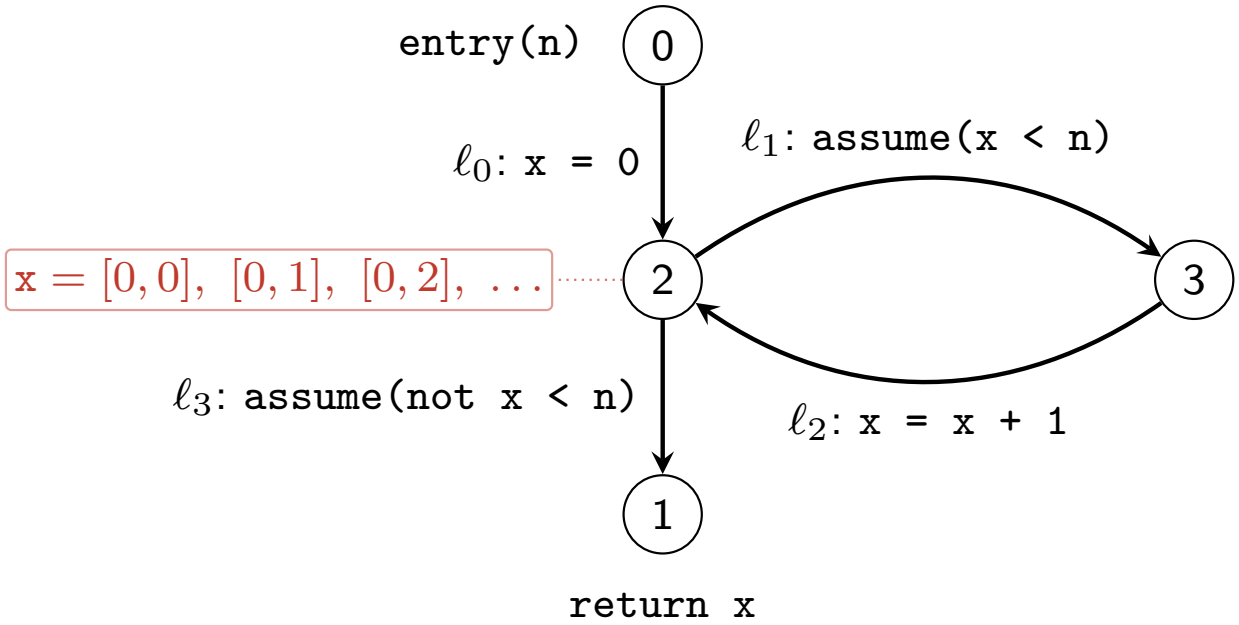
Jihyeok Park




2026 Fall

# Recall

The loop from the previous lecture, in interval:




- Each lap adds one: $[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq \cdots$ Interval has **infinite height**.
- Today: two operators that fix this. **Widening** makes it stop; **narrowing** wins back what widening lost.

# Contents

1. Widening

2. Narrowing

3. Relational Domains

4. Selective Analysis

5. In Python

# Contents

1. Widening

2. Narrowing

3. Relational Domains

4. Selective Analysis

5. In Python

# Widening

If a bound is still moving, **jump** it to infinity:

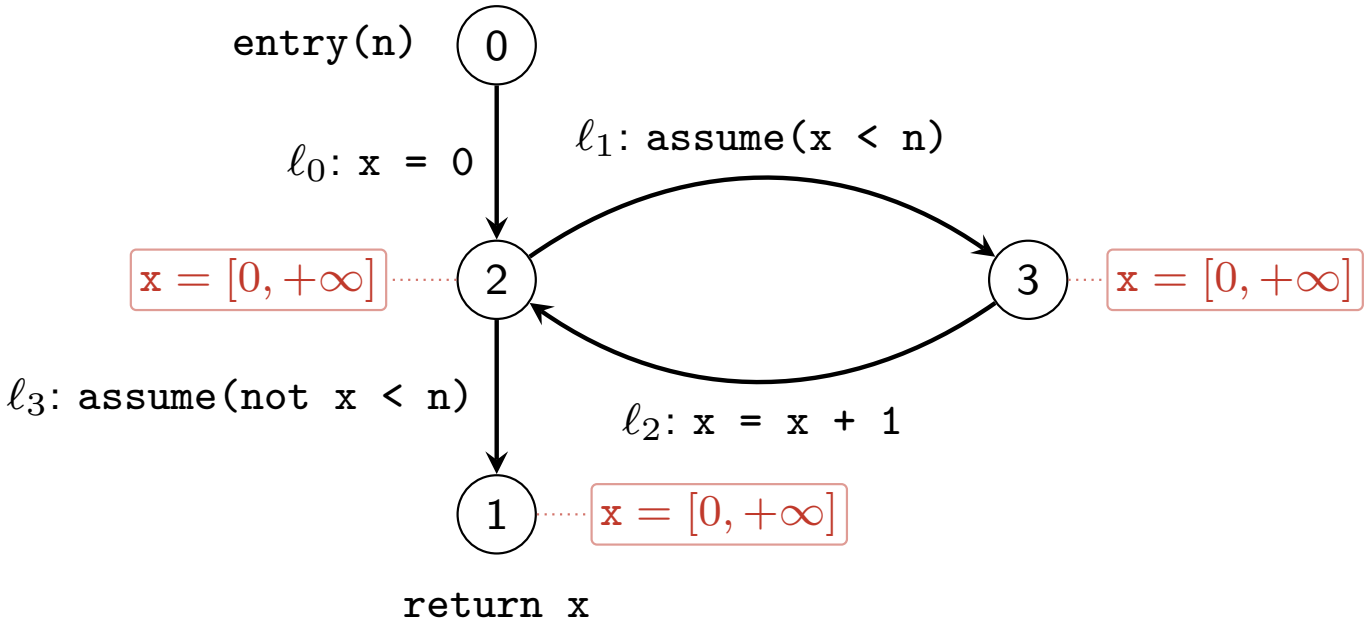
$$[a, b] \triangledown [c, d] = [\text{ if } c < a \text{ then } -\infty \text{ else } a \text{ , if } d > b \text{ then } +\infty \text{ else } b \text{ }]$$

| old | new | old $\triangledown$ new |
|---|---|---|
| $[0, 0]$ | $[0, 1]$ | $[0, +\infty]$ |
| $[0, 5]$ | $[-1, 5]$ | $[-\infty, 5]$ |
| $[0, 5]$ | $[2, 3]$ | $[0, 5]$ |

- It is an **upper bound** of old and new, so the result stays **sound**.
- Each bound can jump only once, so the iteration **stops**.

# Widening, Step by Step

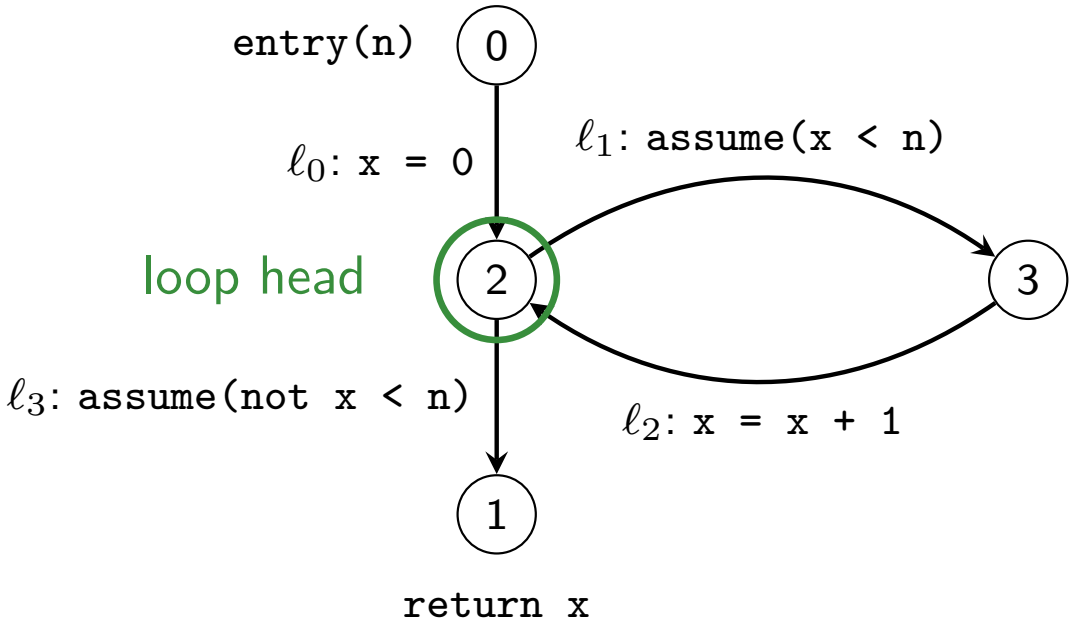
At the loop head, use $\triangledown$ instead of $\sqcup$:




A fixpoint after two laps, not never. The answer $[0, +\infty]$ is right: x can be any $\geq 0$.

# Where to Widen

At the loop head, use $\triangledown$ instead of $\sqcup$:




- Every loop goes through its head. Elsewhere, plain $\sqcup$.

# Contents

1. Widening

2. Narrowing

3. Relational Domains

4. Selective Analysis

5. In Python

# Refinement by Conditions

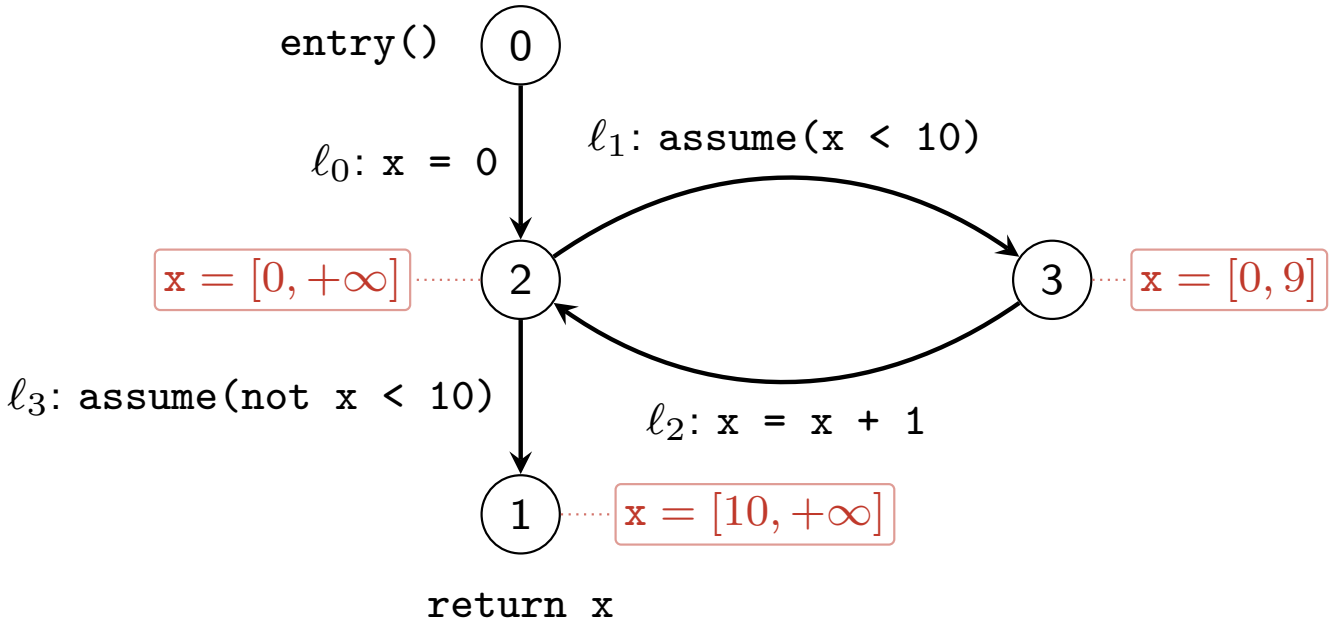
A branch tells us something. On each edge, **meet** with what the condition says:

| before | edge | after |
|---|---|---|
| $\mathtt{x} = [0, 100]$ | `assume(x < 10)` | $[0, 100] \sqcap [-\infty, 9] = [0, 9]$ |
| $\mathtt{x} = [0, 100]$ | `assume(not x < 10)` | $[0, 100] \sqcap [10, +\infty] = [10, 100]$ |
| $\mathtt{x} = [0, 5]$ | `assume(not x < 10)` | $\bot$ (never taken) |

- Two unknown sides work the same way: from `x < y` with $\mathtt{x} = [0, 100]$ and $\mathtt{y} = [0, 50]$, we get $\mathtt{x} = [0, 49]$ and $\mathtt{y} = [1, 50]$.
- This is what `refine` does, and it will save us twice today.

# Precision Lost

The same loop with a constant bound, after widening:




- The body is still exact: `assume(x < 10)` cuts $+\infty$ down to $9$.
- The head and the exit are not. The real values are $[0, 10]$ and $[10, 10]$. Widening jumped to $+\infty$ and never looked back.

# Narrowing

After widening, run the loop **once more**, and only pull back the bounds that went to infinity:

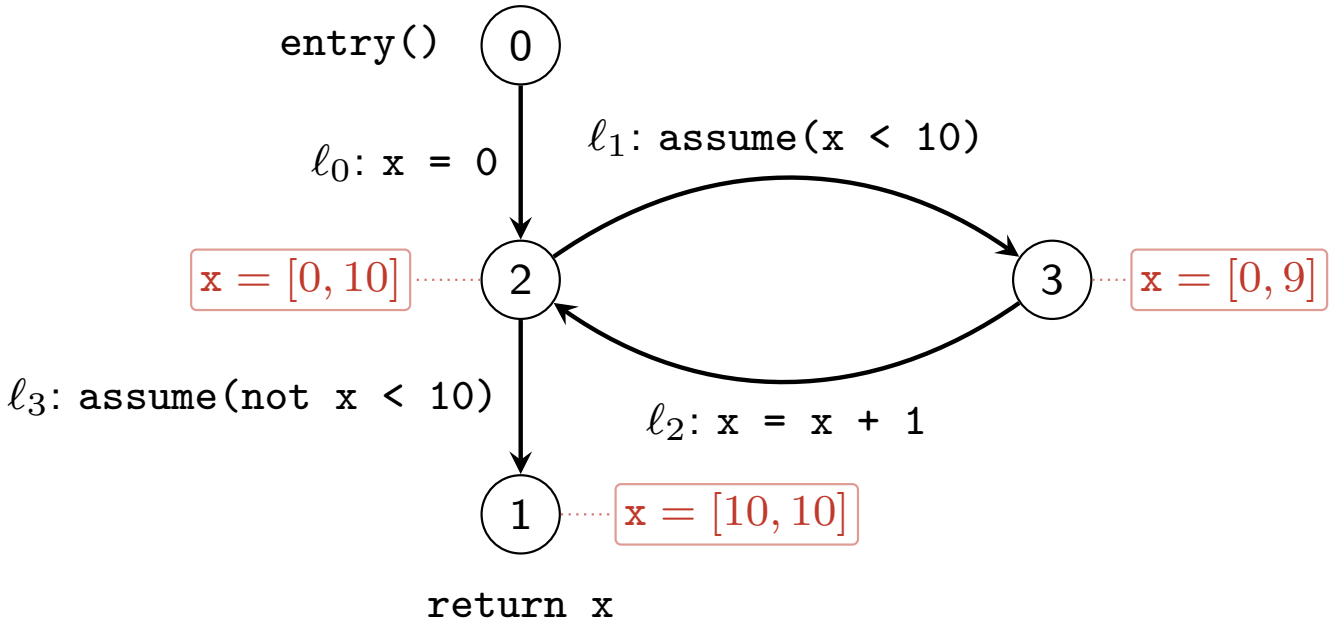
$$[a,b] \triangle [c,d] = [\text{ if } a = -\infty \text{ then } c \text{ else } a \text{ , if } b = +\infty \text{ then } d \text{ else } b \text{ }]$$

| old | new | old $\triangle$ new |
|---|---|---|
| $[0, +\infty]$ | $[0, 10]$ | $[0, 10]$ |
| $[-\infty, 5]$ | $[-3, 4]$ | $[-3, 5]$ |

- Finite bounds stay as they are, so we never go below a sound answer.
- Each infinite bound is pulled back at most once, so it stops.
- Why not just $\sqcap$? It can keep shrinking forever:
  $[-\infty, -1] \sqsupset [-\infty, -2] \sqsupset [-\infty, -4] \sqsupset \cdots$

# Narrowing, Step by Step

Start from the widened result, and go around once more with $\triangle$:




The exit: $[0, 10] \sqcap [10, +\infty] = [10, 10]$. **Exact**.

# Why It Matters

```
def f():
    xs = [0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0]
    x = 0
    while x < 10:
        x = x + 1
    return xs[x]
```

`xs` has 11 elements: indices $0..10$.

| | `x` at the end | `xs[x]` |
|---|---|---|
| widening | $[10, +\infty]$ | alarm |
| $+$ narrowing | $[10, 10]$ | safe |

- The program is safe. With widening alone, the analyzer reports a possible **buffer overflow**: a **false alarm**.
- In C, an out-of-bounds index is a security bug. Proving indices in range is what interval analyzers are used for.

# Delayed Widening

Join $k$ times at the loop head first, then start widening.

| $k$ | head, lap by lap | head | exit |
|---|---|---|---|
| $0$ | $[0,0],\ [0,+\infty]$ | $[0,+\infty]$ | $[10,+\infty]$ |
| $3$ | $[0,0],\ [0,1],\ [0,2],\ [0,3],\ [0,+\infty]$ | $[0,+\infty]$ | $[10,+\infty]$ |
| $10$ | $[0,0],\ [0,1],\ \ldots,\ [0,10]$ | $[0,10]$ | $[10,10]$ |

(the `while x < 10` loop, without narrowing)

- A loop that stops within $k$ laps is analyzed **exactly**.
- A longer loop pays $k$ extra laps for nothing. Small $k$ ($2$ or $3$) is the usual choice.

# Widening with Thresholds

Instead of jumping to $+\infty$, jump to the **next constant** in a set $T$, usually the constants of the program.

| $T$ | head, lap by lap | exit |
|---|---|---|
| $\emptyset$ | $[0,0],\ [0,+\infty]$ | $[10,+\infty]$ |
| $\{0,10\}$ | $[0,0],\ [0,10]$ | $[10,10]$ |
| $\{0,5,10\}$ | $[0,0],\ [0,5],\ [0,10]$ | $[10,10]$ |

- With the right constant, the answer is exact and narrowing is not even needed.
- Each useless constant costs one more lap. In a large program, "every constant" is a lot of laps.

# Contents

1. Widening

2. Narrowing

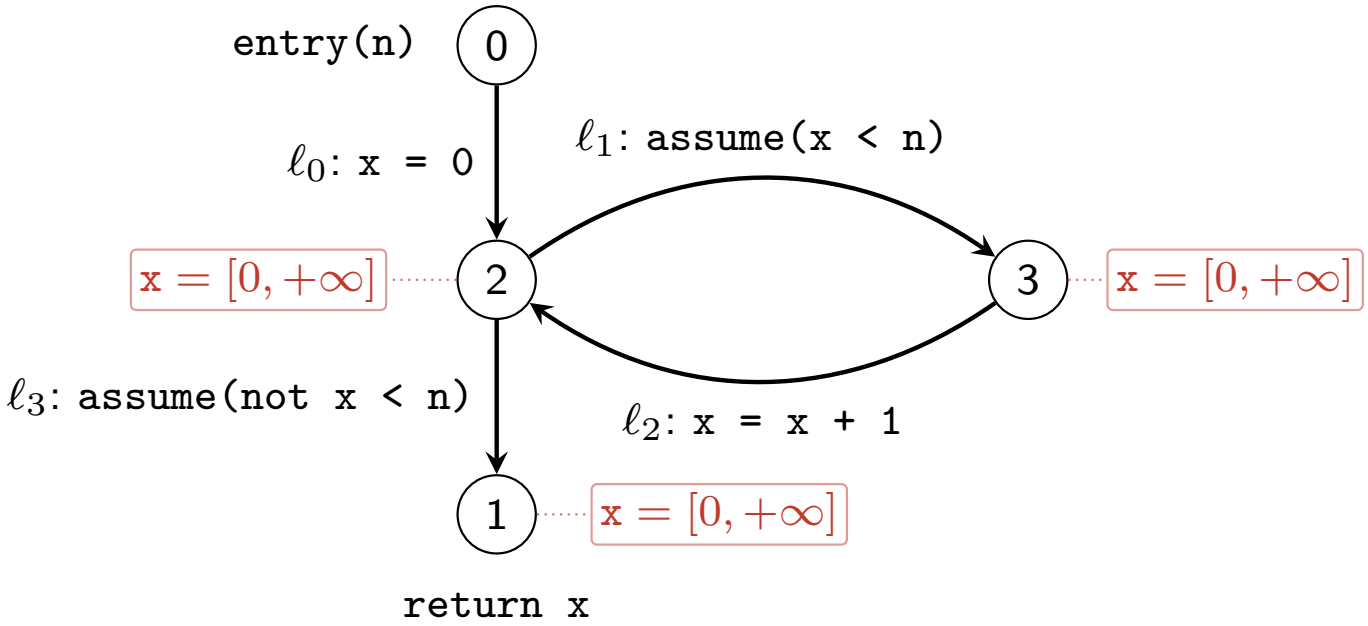
3. Relational Domains

4. Selective Analysis

5. In Python

# When the Bound Is a Variable

Back to the first loop, with everything we have:




- Narrowing and thresholds do not help: the bound is $\mathtt{n}$, not a number, and $\mathtt{n}$ can be anything.
- What is true: $\mathtt{x} < \mathtt{n}$ in the body, and $\mathtt{x} \geq \mathtt{n}$ at the exit. Not a range of $\mathtt{x}$, but a **relation** between $\mathtt{x}$ and $\mathtt{n}$.

# What Intervals Cannot Say

The same problem in a bounds check. `s` is the sum of the window `xs[j], ..., xs[i - 1]`: at most 3 elements, fewer if the sum is over 10.

```
def f(xs, n):  # n elements
    i = 0
    j = 0
    s = 0
    while i < n:
        s = s + xs[i]       # safe?
        i = i + 1
        if j + 3 < i or 10 < s:
            s = s + -xs[j] # safe?
            j = j + 1
    return s
```

Interval says

$$\texttt{i} = [0, +\infty] \quad \texttt{j} = [0, +\infty]$$

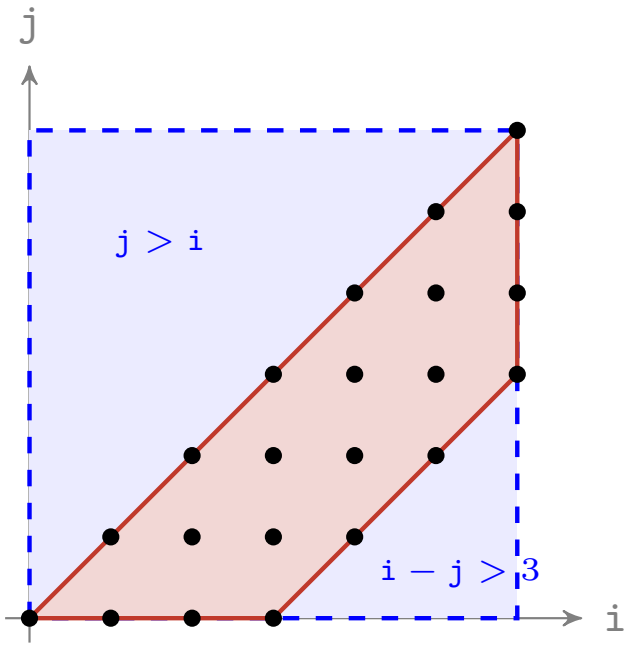
and `n` is unknown.

Is `i < n` at `xs[i]`?
Is `j < n` at `xs[j]`?
Cannot tell.

- Both are safe: $\texttt{i} < \texttt{n}$ by the loop condition, and $\texttt{j} < \texttt{i} \leq \texttt{n}$. But these are **relations**, and interval is **non-relational**.

# Octagons

Keep bounds on $\pm x \pm y$ too: constraints $\pm x \pm y \leq c$.




The pairs $(\mathtt{i}, \mathtt{j})$ at the loop head of `f`, for $\mathtt{n} = 6$:

| | interval | octagon |
|---|---|---|
| $\mathtt{i}$ | $[0, 6]$ | $[0, 6]$ |
| $\mathtt{j}$ | $[0, 6]$ | $[0, 6]$ |
| $\mathtt{i} + \mathtt{j}$ | | $[0, 12]$ |
| $\mathtt{i} - \mathtt{j}$ | | $[0, 3]$ |

- An octagon is a **table of bounds** on $x$, $y$, $x + y$, $x - y$. Each bound is an edge: up to 8 edges, hence the name.
- Here the two bounds on $\mathtt{i} - \mathtt{j}$ cut off two corners of the box.

# The Octagon Domain

The **forms**: $\pm x$ and $\pm x \pm y$, for variables $x \neq y$.

- An element $o$ is $\bot$, or an **upper bound** for every form: $o(e) \in \mathbb{Z} \cup \{+\infty\}$.
- $\gamma(o) = \{\sigma \mid e(\sigma) \leq o(e) \text{ for every form } e\}$.
- Order, join, meet: **form by form**, like intervals.

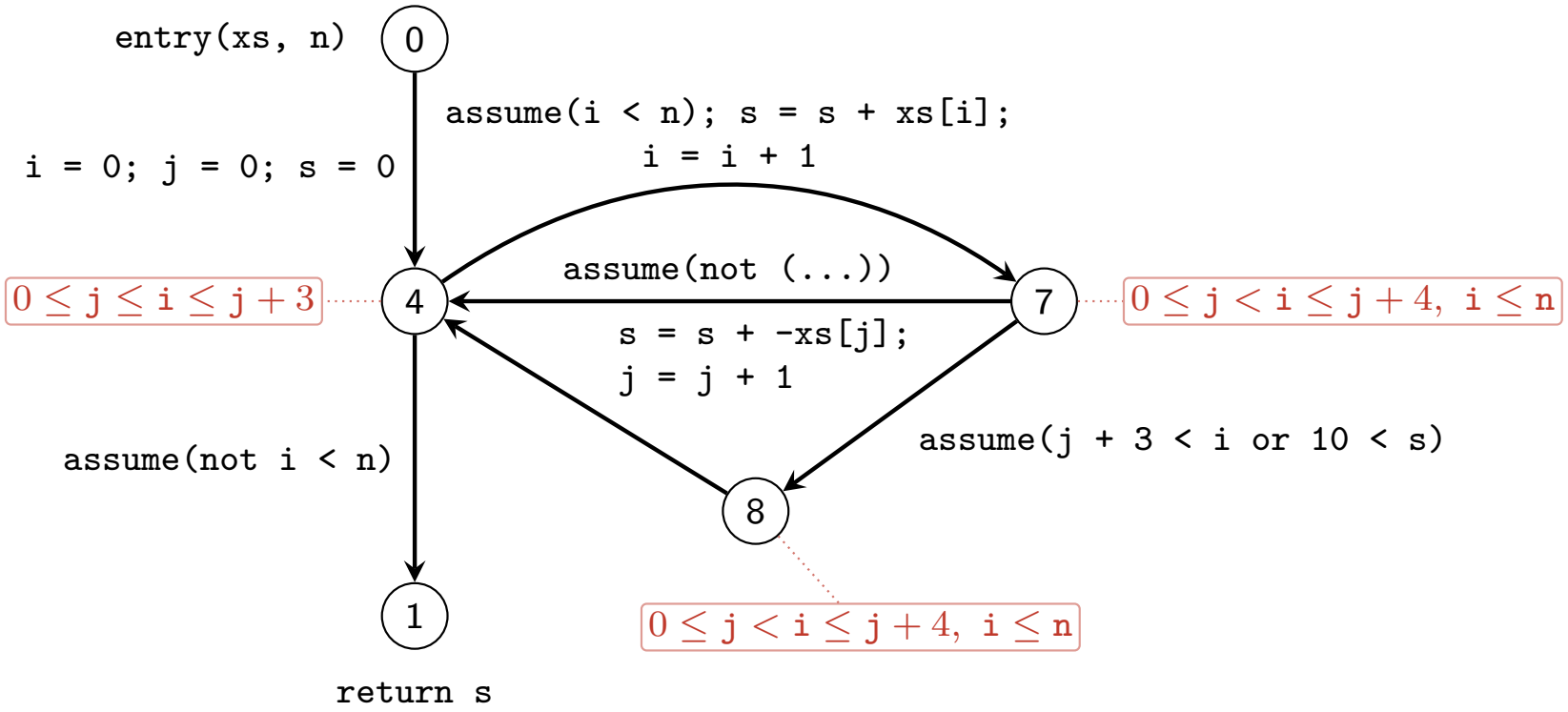
$$o_1 \sqsubseteq o_2 \iff \forall e.\ o_1(e) \leq o_2(e) \qquad (o_1 \sqcup o_2)(e) = \max(o_1(e), o_2(e))$$

- Widening: keep a bound only if it did not grow.

$$(o_1 \triangledown o_2)(e) = \text{if } o_2(e) \leq o_1(e) \text{ then } o_1(e) \text{ else } +\infty$$

- **Closure**, before $\sqsubseteq$ and $\sqcup$: tighten each bound through the others, e.g. $o(x - z) \leq o(x - y) + o(y - z)$.

# Octagon, Step by Step

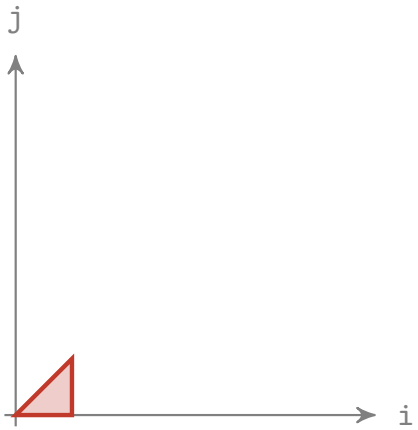



No change at 4. At 8, **closure** adds $j < i$ and $i \le n$ into $j < n$: with $0 \le j$, `xs[j]` is **safe**. Interval says $j = [0, +\infty]$: an alarm.

# The Loop Head, as a Table

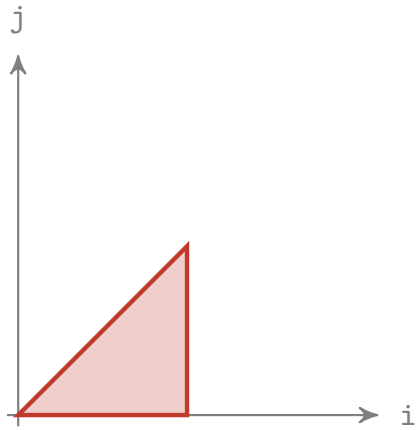
What the analyzer stores at point 4, for `i` and `j`. A row $[a, b]$ for `i` $-$ `j` is two forms: $o(\texttt{i} - \texttt{j}) = b$, $o(\texttt{j} - \texttt{i}) = -a$.

lap 1, join



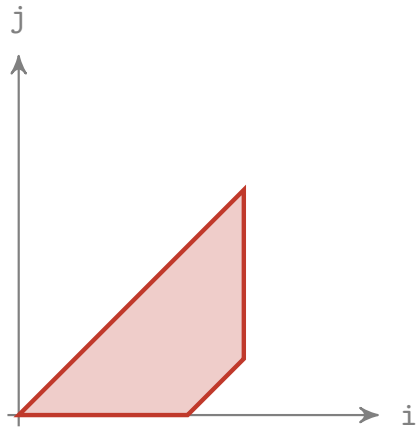

| | |
|---|---|
| `i` | $[0, 1]$ |
| `j` | $[0, 1]$ |
| `i` + `j` | $[0, 2]$ |
| `i` $-$ `j` | $[0, 1]$ |

lap 3, join



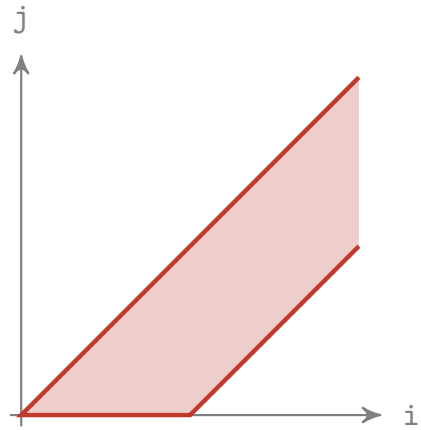

| | |
|---|---|
| `i` | $[0, 3]$ |
| `j` | $[0, 3]$ |
| `i` + `j` | $[0, 6]$ |
| `i` $-$ `j` | $[0, 3]$ |

lap 4, join




| | |
|---|---|
| `i` | $[0, 4]$ |
| `j` | $[0, 4]$ |
| `i` + `j` | $[0, 8]$ |
| `i` $-$ `j` | $[0, 3]$ |

lap 4, widen




| | |
|---|---|
| `i` | $[0, +\infty]$ |
| `j` | $[0, +\infty]$ |
| `i` + `j` | $[0, +\infty]$ |
| `i` $-$ `j` | $[0, 3]$ |

- At lap 4 the window is full: the row `i` $-$ `j` stays $[0, 3]$.
- Widening sends the rows that moved to $+\infty$ and keeps `i` $-$ `j`: the band of the first picture.

# The Dial

| domain | constraints | proves | cost |
|---|---|---|---|
| interval | $c \le x \le d$ | `xs[x]` after `while x < 10` | $O(n)$ |
| octagon | $\pm x \pm y \le c$ | `xs[j]` with `j` $<$ `i` $\le$ `n` | $O(n^3)$ |
| polyhedra | $\sum a_k x_k \le c$ | also `j` $=$ 2`i` | exponential |

($n$: the number of variables)

- The same dial as before: more precise, more expensive.
- Real analyzers mix them. Astrée runs octagons, but only on small groups of variables. The Apron library offers all three.
- Which groups? Which constants for thresholds? **Next**.

# Contents

1. Widening

2. Narrowing

3. Relational Domains

4. Selective Analysis

5. In Python

# Which Constants?

Thresholds help only if they include the **right** constant.

```
i = 0;
while (i != 4) {
  i = i + 1;
  assert(i <= 4);
}
```

| $T$ | `i` at the head | `assert` |
|---|---|---|
| $\emptyset$ | $[0, +\infty]$ | alarm |
| $\{4\}$ | $[0, 4]$ | proved |

Narrowing cannot help: `i != 4` says nothing about $[0, +\infty]$.

Using every constant works, but is **slow**. The idea: **learn which constants are useful** from programs we have already analyzed.

- **Training**: on many programs, run once with every constant. Note which ones actually became bounds, and what they look like (in a condition? an array size? $2^k$?).
- **Use**: on a new program, keep only the constants that look like that.

# Which Variables?

Octagon over $n$ variables costs $O(n^3)$. Real analyzers split the variables into small **packs**.

```
int a = b;
int c = input();
for (i = 0; i < b; i++) {
  assert(i < a); // Query 1
  assert(i < c); // Query 2
}
```

- Query 1 needs $\mathtt{a} = \mathtt{b}$ and $\mathtt{i} < \mathtt{b}$.
- Query 2 is **never** provable: $\mathtt{c}$ is user input.
- So packs $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and $\{\mathtt{c}\}$.

- Choose the packs by a cheap **pre-analysis** ("could octagon prove this?"), or **learn** to predict its answer.
- The same idea as thresholds: spend precision only **where it pays**.

# Contents

1. Widening

2. Narrowing

3. Relational Domains

4. Selective Analysis

5. In Python

# The Engine, with Widening

```
def analyze(g, dom, widening=True, narrowing=False, delay=0, ...):
    ...
    while work:
        n = work.pop()
        joined = incoming(dom, g, n, inn)
        # widen only at loop heads, after `delay` plain joins
        widen_here = widening and n in g.loop_heads and seen[n] > delay
        new = st_widen(dom, inn[n], joined) if widen_here else joined
        seen[n] += 1
        ...
    if narrowing:
        ...          # st_narrow: pull infinite bounds back
    return inn
```

- A general worklist for any abstract domain. At loop heads, $\sqcup$ becomes $\nabla$ after `delay` plain joins.
- Thresholds are not here: they live in the domain. `Interval(T).widen` picks the next one; the engine just calls it.

# Running It

Once your `interval.py` is done:

```
>>> from domain import analyze
>>> from interval import Interval, Itv

>>> print(Interval().widen(Itv(0, 0), Itv(0, 1)))
[0, inf]
>>> print(Interval({0, 10}).widen(Itv(0, 0), Itv(0, 1)))
[0, 10]

>>> src = """
... def f():
...     x = 0
...     while x < 10:
...         x = x + 1
...     return x
... """
>>> g = build(parse(src))
>>> print(analyze(g, Interval())[2]["x"])
[0, inf]
>>> print(analyze(g, Interval({0, 10}))[2]["x"])
[0, 10]
```

# Homework 2

- Fill in `interval.py`: the interval domain, with widening (thresholds too) and narrowing.
- Everything you need, and the tests that grade you:

  **github.com/ku-plrg-classroom/py-interval**
- Due November 9 on the **LMS**. Submit `interval.py` only.

# Summary

1. Widening
2. Narrowing
3. Relational Domains
4. Selective Analysis
5. In Python

# Next Lecture

- Pointer Analysis

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr