Software Verification

Introduction to Abstract Interpretation

Ahmed Rezine

IDA, Linköpings Universitet

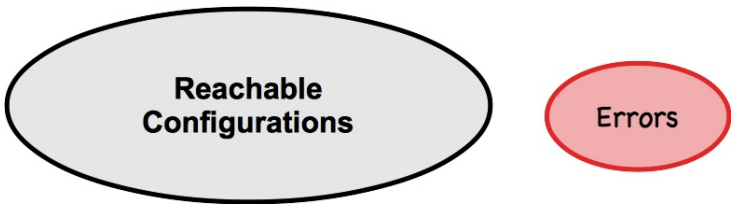
Spring 2025

Outline

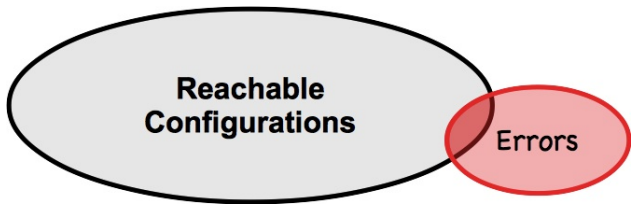
Outline

Verification

- We want to answer whether some program behaves correctly. We define "correctness" soon.
- For now, assume that means some erroneous configurations are not reachable
- We say the program is **safe**



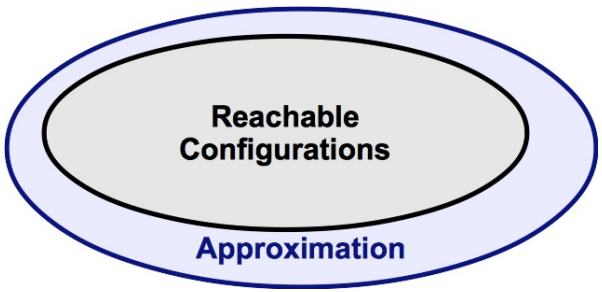
Safe Program



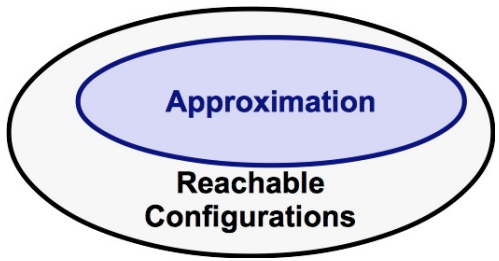
Unsafe Program

Verification problem and approximations

- The idea is then to come up with efficient approximations to give correct answers in as many cases as possible.



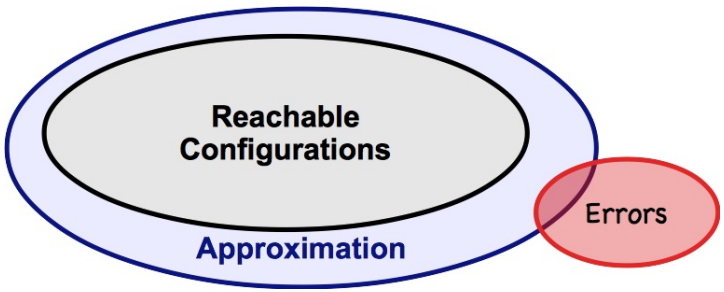
Over-approximation



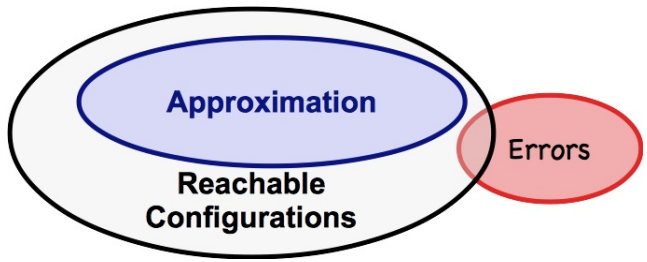
Under-approximation

Program verification and the price of approximations

- A sound analysis cannot give **false negatives**
- A complete analysis cannot give **false positives**



False Positive



False Negative

Approaches

- We discussed:
  - Explicit/symbolic model checking
  - Symbolic Execution relying on SMT solvers
  - Deductive frameworks with Hoare triples
- Today: a systematic and automatic framework for generating over-approximations

Outline

A simple language

Assume a simple language over a finite number of integer variables ranging over $\mathbb{Z}$.

- `exp ::= n | x | exp + exp | exp - exp | *`
- `cond ::= true | false | exp >= exp | cond && cond`
- `cmd ::= x:= exp | if cond goto label | assert cond | end`

```
L1. x:= *
L2. y:= x
L3. x:= x + y
L4. if x >= y goto L6
L5. end
L6. assert y >= 0
L7. end
```

```
L1. x:= 0
L2. y:= 0
L3. x:= x + 1
L4. y:= y + 2
L5. if x < 100 goto L3
L6. assert y = 200
L7. end
```

We want to generate "invariants", e.g., characterize “facts” that hold at each line

Outline

Abstract Interpretation

- In the concrete semantics, one can associate to each label a set of possible mappings from the variables to their values.
- Concrete semantics is precise: it only captures possible states. It is however too inefficient to be feasible in practice (loops, arbitrary inputs, arrays, recursion, concurrent interleavings, etc)
- Instead, efficiently and soundly over-approximate while tracking “facts” of certain “forms”

Abstract Interpretation

Idea:

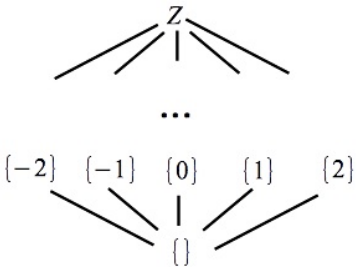
- Assume an "abstract domain" fixing the "form" of the tracked facts (e.g., $a \leq x \leq b$ or $a \leq x - y \leq b$ or ...)
- Define an abstract semantics that over-approximates the concrete semantics (e.g., abstract additions, substractions, comparisons, etc)
- Iterate computation with abstract semantics untill fixpoint. Deduce facts about the original semantics.

A simple abstract domain: the sign example

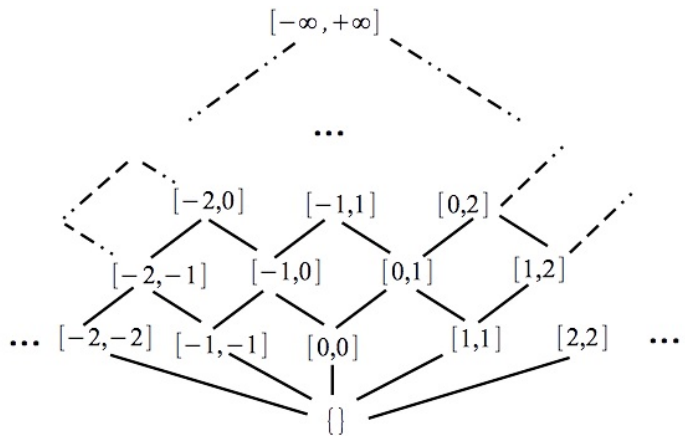
- For an integer variable, the set of concrete values at a location is in $\mathcal{P}(\mathbb{Z})$. Concrete sets of variables' values can be ordered with the subset relation $\sqsubseteq_c$ on $\mathcal{P}(\mathbb{Z})$. We write $S_1 \sqsubseteq_c S_2$ to mean that S_1 is more precise than S_2.
- If you are only interested in the signs of variables' values, you can associate, at each label and to each variable, a subset of $\{-, 0, +\}$.
- We approximate concrete values with an element in $\mathcal{P}(\{-, 0, +\})$. For instance, $\{+\}$ reflects the variable is larger than zero. For A_1, A_2 in $\mathcal{P}(\{-, 0, +\})$, we write $A_1 \sqsubseteq_a A_2$ to mean that A_1 is more precise than A_2.

Lattices

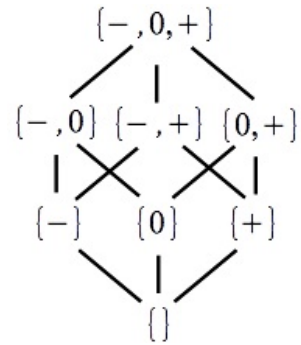
- A lattice is a poset $(Q, \sqsubseteq)$ where each pair p, q in Q has:
  - a greatest lower bound (aka. meet) $p \sqcap q$ wrt. $\sqsubseteq$, and
  - a least upper bound (aka. join) $p \sqcup q$ wrt. $\sqsubseteq$
- If S is a set, then $(\mathcal{P}(S), \subseteq, \bot, \top, \cup, \cap)$ is a complete lattice: i.e, a lattice where each subset has a least upper bound and a greatest lower bound.
- $(\mathcal{P}(\mathbb{Z}), \sqsubseteq_c)$ (resp. $(\mathcal{P}(\{-, 0, +\}), \sqsubseteq_a)$) is a complete lattices where $\top_c = \mathbb{Z}$ and $\bot_c = \{\}$ (resp. $\top_a = \{-, 0, +\}$ and $\bot_a = \{\}$)



subsets



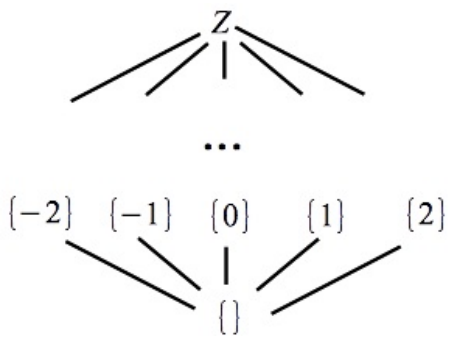
intervals



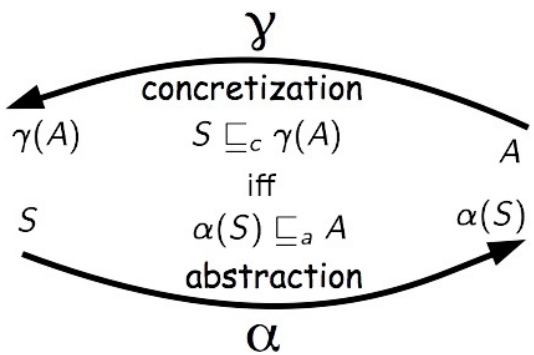
signs

The sign example: abstraction and concretization

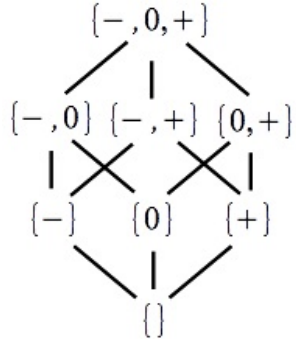
- An abstraction function: $\alpha(\{1, 10, 100, 103, 2021\}) = \{+\}$, $\alpha(\{-10, -3, 10\}) = \{-, +\}$, $\alpha(\{-3, 0\}) = \{-, 0\}$
- A concretization function: $\gamma(\{0\}) = \{0\}$, $\gamma(\{-, 0\}) = \{v | v \leq 0\}$ and $\gamma(\{+\}) = \{v | v > 0\}$



Concrete lattice
$(\mathcal{P}(\mathbb{Z}), \sqsubseteq_c)$



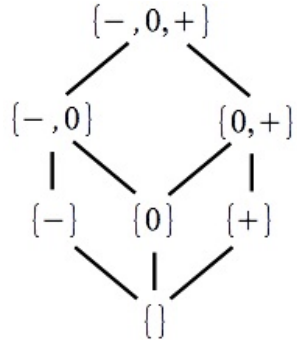
A Galois connection
(α, γ)



Abstract lattice
$(\mathcal{P}(\{-, 0, +\}), \sqsubseteq_a)$

Fixpoint computation: simple example (1/6)

```
L1. x:= *
L2. y:= x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:
//y:
L1. x:= *
//x:
//y:
L2. y:= x
//x:
//y:
L3. if x > 0 goto L5
//x:
//y:
L4. end
//x:
//y:
L5. assert x >= 0
//x:
//y:
L6. end
```

→

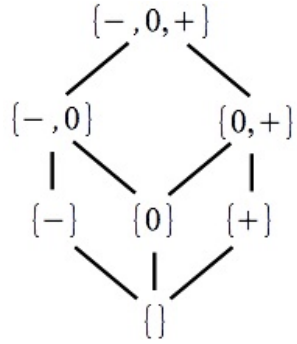
```
//x:⊤
//y:⊤
L1. x:= *
//x:
//y:
L2. y:= x
//x:
//y:
L3. if x > 0 goto L5
//x:
//y:
L4. end
//x:
//y:
L5. assert x >= 0
//x:
//y:
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x:= *
//x:⊥
//y:⊥
L2. y:= x
//x:⊥
//y:⊥
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

Fixpoint computation: simple example (2/6)

```
L1. x:= *
L2. y:= x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:⊤
//y:⊤
L1. x:= *
//x:⊥
//y:⊥
L2. y:= x
//x:⊥
//y:⊥
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

→

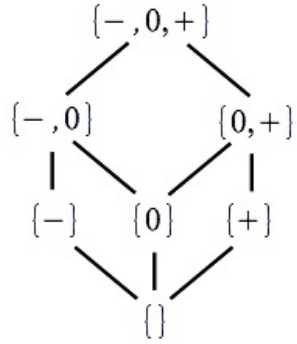
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊥⊔⊤
//y:⊥⊔⊤
L2. y:= x
//x:⊥⊔⊥
//y:⊥⊔⊥
L3. if x > 0 goto L5
//x:⊥⊔(⊥ ⊓ {−,0})
//y:⊥⊔(⊥ ⊓ ⊤)
L4. end
//x:⊥⊔(⊥ ⊓ {+})
//y:⊥⊔(⊥ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔(⊥ ⊓ {0,+})
//y:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x:= *
//x:⊤
//y:⊤
L2. y:= x
//x:⊥
//y:⊥
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

Fixpoint computation: simple example (3/6)

```
L1. x:= *
L2. y:= x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:⊤
//y:⊤
L1. x:= *
//x:⊤
//y:⊤
L2. y:= x
//x:⊥
//y:⊥
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

→

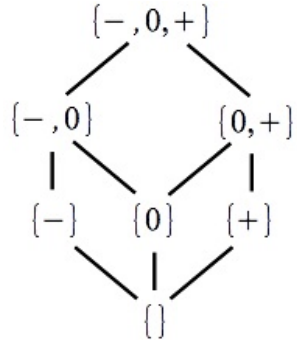
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y:= x
//x:⊥⊔⊤
//y:⊥⊔⊤
L3. if x > 0 goto L5
//x:⊥⊔(⊥ ⊓ {−,0})
//y:⊥⊔(⊥ ⊓ ⊤)
L4. end
//x:⊥⊔(⊥ ⊓ {+})
//y:⊥⊔(⊥ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔(⊥ ⊓ {0,+})
//y:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x:= *
//x:⊤
//y:⊤
L2. y:= x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

Fixpoint computation: simple example (4/6)

```
L1. x := *
L2. y := x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:⊥
//y:⊥
L4. end
//x:⊥
//y:⊥
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

→

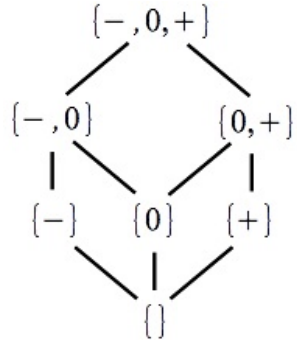
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x := *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y := x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:⊥⊔(⊤ ⊓ {−,0})
//y:⊥⊔(⊤ ⊓ ⊤)
L4. end
//x:⊥⊔(⊤ ⊓ {+})
//y:⊥⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔(⊥ ⊓ {0,+})
//y:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:{−,0}
//y:⊤
L4. end
//x:{+}
//y:⊤
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

Fixpoint computation: simple (5/6)

```
L1. x := *
L2. y := x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:{−,0}
//y:⊤
L4. end
//x:{+}
//y:⊤
L5. assert x >= 0
//x:⊥
//y:⊥
L6. end
```

→

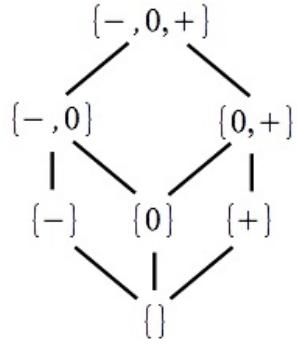
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x := *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y := x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
//y:⊤⊔(⊤ ⊓ ⊤)
L4. end
//x:{+}⊔(⊤ ⊓ {+})
//y:⊤⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔({+} ⊓ {0,+})
//y:⊥⊔(⊤ ⊓ ⊤)
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:{−,0}
//y:⊤
L4. end
//x:{+}
//y:⊤
L5. assert x >= 0
//x:{+}
//y:⊤
L6. end
```

Fixpoint computation: simple (6/6)

```
L1. x := *
L2. y := x
L3. if x > 0 goto L5
L4. end
L5. assert x >= 0
L6. end
```




```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:{−,0}
//y:⊤
L4. end
//x:{+}
//y:⊤
L5. assert x >= 0
//x:{+}
//y:⊤
L6. end
```

→

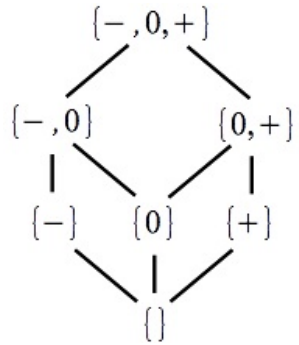
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x := *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y := x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
//y:⊤⊔(⊤ ⊓ ⊤)
L4. end
//x:{+}⊔(⊤ ⊓ {+})
//y:⊤⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:{+}⊔({+} ⊓ {0,+})
//y:⊤⊔(⊤ ⊓ ⊤)
L6. end
```

→

```
//x:⊤
//y:⊤
L1. x := *
//x:⊤
//y:⊤
L2. y := x
//x:⊤
//y:⊤
L3. if x > 0 goto L5
//x:{−,0}
//y:⊤
L4. end
//x:{+}
//y:⊤
L5. assert x >= 0
//x:{+}
//y:⊤
L6. end
```

Fixpoint computation: another without $\neq 0$ (1/5)

```
L1. x := *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x := *
//x:⊥
L2. if x > 0 goto L5
//x:⊥
L3. if x < 0 goto L5
//x:⊥
L4. end
//x:⊥
L5. assert x != 0
//x:⊥
L6. end
```

→

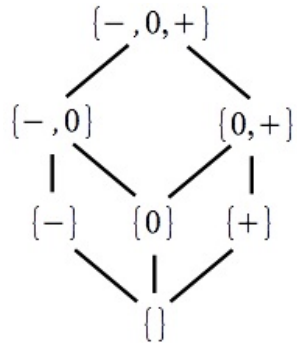
```
//x:⊤,
L1. x := *
//x:⊥⊔⊤
L2. if x > 0 goto L5
//x:⊥⊔(⊥ ⊓ {−,0})
L3. if x < 0 goto L5
//x:⊥⊔(⊥ ⊓ {0,+})
L4. end
//x:⊥⊔(⊥ ⊓ {+}) ⊔ (⊥ ⊓ {−})
L5. assert x != 0
//x:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:⊥
L3. if x < 0 goto L5
//x:⊥
L4. end
//x:⊥
L5. assert x != 0
//x:⊥
L6. end
```

Fixpoint computation: another without $\neq 0$ (2/5)

```
L1. x := *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:⊥
L3. if x < 0 goto L5
//x:⊥
L4. end
//x:⊥
L5. assert x != 0
//x:⊥
L6. end
```

→

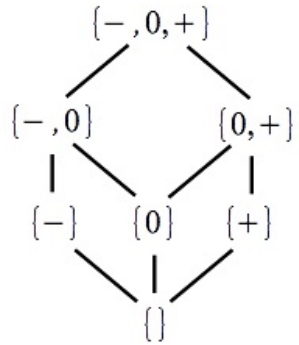
```
//x:⊤,
L1. x := *
//x:⊤⊔⊤
L2. if x > 0 goto L5
//x:⊥⊔(⊤ ⊓ {−,0})
L3. if x < 0 goto L5
//x:⊥⊔(⊥ ⊓ {0,+})
L4. end
//x:⊥⊔(⊤ ⊓ {+}) ⊔ (⊥ ⊓ {−})
L5. assert x != 0
//x:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:⊥
L4. end
//x:{+}
L5. assert x != 0
//x:⊥
L6. end
```

Fixpoint computation: another without $\neq 0$ (3/5)

```
L1. x:= *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x:= *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:⊥
L4. end
//x:{+}
L5. assert x != 0
//x:⊥
L6. end
```

→

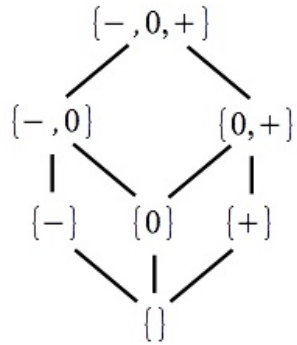
```
//x:⊤,
L1. x:= *
//x:⊤⊔⊤
L2. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
L3. if x < 0 goto L5
//x:⊥⊔({−,0} ⊓ {0,+})
L4. end
//x:{+}⊔(⊤ ⊓ {+}) ⊔ ({−,0} ⊓ {−})
L5. assert x != 0
//x:{+}⊔({+} ⊓ ⊤)
L6. end
```

→

```
//x:⊤,
L1. x:= *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:⊤
L5. assert x != 0
//x:{+}
L6. end
```

Fixpoint computation: another without $\neq 0$ (4/5)

```
L1. x := *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:{+}
L5. assert x != 0
//x:{+}
L6. end
```

→

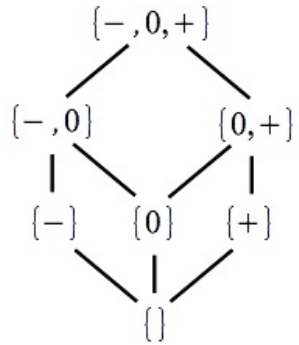
```
//x:⊤,
L1. x := *
//x:⊤⊔⊤
L2. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
L3. if x < 0 goto L5
//x:{0}⊔({−,0} ⊓ {0,+})
L4. end
//x:{+}⊔(⊤ ⊓ {+}) ⊔ ({−,0} ⊓ {−})
L5. assert x != 0
//x:{+}⊔({+} ⊓ ⊤)
L6. end
```

→

```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:⊤
L5. assert x != 0
//x:{+}
L6. end
```

Fixpoint computation: another without $\neq 0$ (5/5)

```
L1. x := *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:⊤
L5. assert x != 0
//x:{+}
L6. end
```

$\longrightarrow$

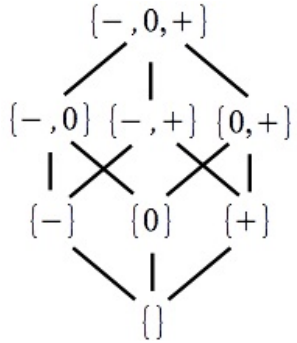
```
//x:⊤,
L1. x := *
//x:⊤⊔⊤
L2. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
L3. if x < 0 goto L5
//x:{0}⊔({−,0} ⊓ {0,+})
L4. end
//x:⊤⊔(⊤ ⊓ {+}) ⊔ ({−,0} ⊓ {−})
L5. assert x != 0
//x:{+}⊔(⊤ ⊓ ⊤)
L6. end
```

$\longrightarrow$

```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:⊤
L5. assert x != 0
//x:⊤
L6. end
```

Fixpoint computation: same with $\neq 0$

```
L1. x := *
L2. if x > 0 goto L5
L3. if x < 0 goto L5
L4. end
L5. assert x != 0
L6. end
```




```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:{−,+}
L5. assert x != 0
//x:{−,+}
L6. end
```

→

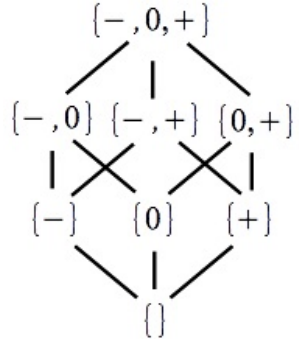
```
//x:⊤,
L1. x := *
//x:⊤⊔⊤
L2. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
L3. if x < 0 goto L5
//x:{0}⊔({−,0} ⊓ {0,+})
L4. end
//x:{−,+}⊔(⊤ ⊓ {+}) ⊔ ({−,0} ⊓ {−})
L5. assert x != 0
//x:{−,+}⊔({−,+} ⊓ {−,+})
L6. end
```

→

```
//x:⊤,
L1. x := *
//x:⊤
L2. if x > 0 goto L5
//x:{−,0}
L3. if x < 0 goto L5
//x:{0}
L4. end
//x:{−,+}
L5. assert x != 0
//x:{−,+}
L6. end
```

Fixpoint computation: signs vs. intervals

```
L1. x:= 0
L2. x:= x + 1
L3. x:= x + 2
L4. assert x >= 2
L5. end
```

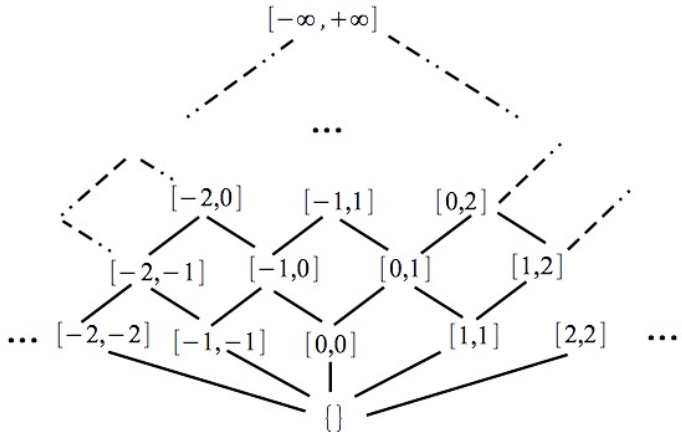



```
//x:⊤
L1. x:= 0
//x:{0}
L2. x:= x + 1
//x:{+}
L3. x:= x + 2
//x:{+}
L4. assert x >= 2
//x:{+}
L5. end
```

```
//x:⊤⊔⊥
L1. x:= 0
//x:{0}⊔{0}
L2. x:= x + 1
//x:{+}⊔{+}
L3. x:= x + 2
//x:{+}⊔{+}
L4. assert x >= 2
//x:{+}⊔{+}
L5. end
```

```
//x:⊤
L1. x:= 0
//x:{0}
L2. x:= x + 1
//x:{+}
L3. x:= x + 2
//x:{+}
L4. assert x >= 2
//x:{+}
L5. end
```

Fixpoint computation: signs vs. intervals



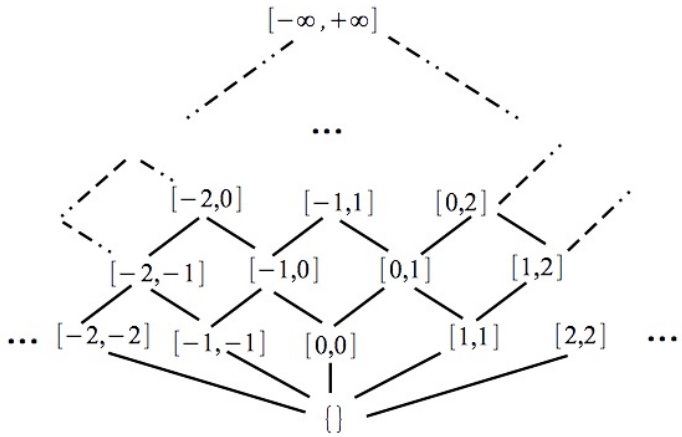
$[a,b] \sqsubseteq [c,d]$ iff $c \leq a$ and $b \leq d$
$[a,b] \sqcup [c,d]$ is $[inf\{a,c\}, sup\{b,d\}]$
$[a,b] \sqcap [c,d]$ is $[sup\{a,c\}, inf\{b,d\}]$

```
//x:⊤
L1. x:= 0
//x:[0,0]
L2. x:= x + 1
//x:[1,1]
L3. x:= x + 2
//x:[3,3]
L4. assert x >= 2
//x:[3,3]
L5. end
```

```
//x:⊤
L1. x:= 0
//x:[0,0]⊔[0,0]
L2. x:= x + 1
//x:[1,1]⊔[1,1]
L3. x:= x + 2
//x:[3,3]⊔[3,3]
L4. assert x >= 2
//x:[3,3]⊔([3,3] ⊓ [2,+oo])
L5. end
```

```
//x:⊤
L1. x:= 0
//x:[0,0]
L2. x:= x + 1
//x:[1,1]
L3. x:= x + 2
//x:[3,3]
L4. assert x >= 2
//x:[3,3]
L5. end
```

Fixpoint computation: loops with intervals



$[a,b] \sqsubseteq [c,d]$ iff $c \leq a$ and $b \leq d$
$[a,b] \sqcup [c,d]$ is $[inf\{a,c\}, sup\{b,d\}]$
$[a,b] \sqcap [c,d]$ is $[sup\{a,c\}, inf\{b,d\}]$

```
//x:⊤
L1. x:= 0
//x:[0,25]
L2. x:= x + 1
//x:[1,26]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

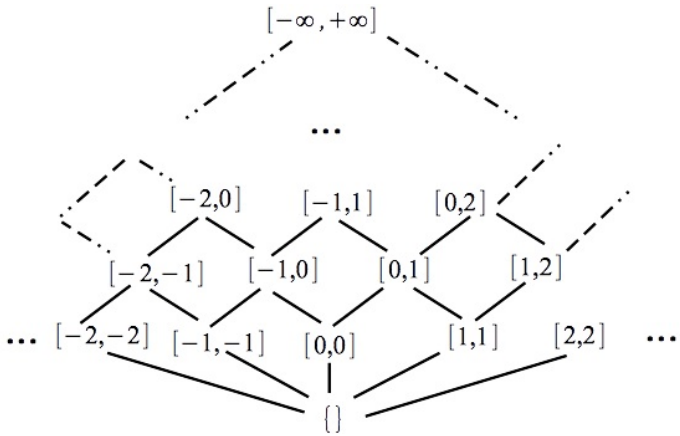
→

```
//x:⊤
L1. x:= 0
//x:[0,25]⊔[0,0]
//   ⊔([1,26] ⊓ [−oo,99])
L2. x:= x + 1
//x:[1,26]⊔[1,26]
L3. if x < 100 goto L2
//x:⊥⊔([1,26] ⊓ [100,+oo])
L4. assert x >= 100
//x:⊥⊔(⊥ ⊓ [100,+oo])
L5. end
```

→

```
//x:⊤
L1. x:= 0
//x:[0,26]
L2. x:= x + 1
//x:[1,26]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

Fixed point computation: widening



$[0,0], [0,1], [0,2], [0,3]$....
would take long time to converge.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:⊤
L1. x:= 0
//x:[0,25]

L2. x:= x + 1
//x:[1,26]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:([0,25]⊔[0,0])
// ⊔([1,26]⊓[-oo,99])
L2. x:= x + 1
//x:([1,26]⊔[1,26])
L3. if x < 100 goto L2
//x:⊥⊔([1,26]⊓[100,+oo])
L4. assert x >= 100
//x:⊥⊔(⊥⊓[100,+oo])
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:([0,25]∇[0,0])
// ∇([1,26]⊓[-oo,99])
L2. x:= x + 1
//x:([1,26]∇[1,26])
L3. if x < 100 goto L2
//x:⊥∇([1,26]⊓[100,+oo]
L4. assert x >= 100
//x:⊥∇(⊥⊓[100,+oo])
L5. end
```

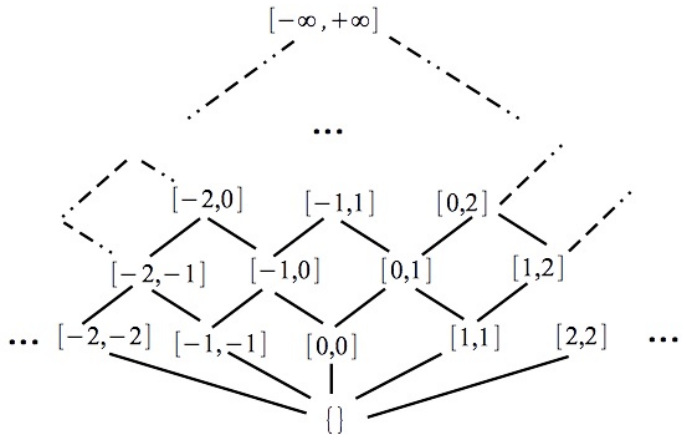
Fixpoint computation: widening

- A widening operator ∇ guarantees termination even in the presence of an infinite-height abstract lattice.
- Conditions on a widening operator ∇
  - for any abstract elements $A^\sharp, B^\sharp$, $(A^\sharp \sqcup_a B^\sharp) \sqsubseteq_a (A^\sharp \nabla B^\sharp)$
  - For any ${C_0}^\sharp \sqcup_a {C_1}^\sharp \sqcup_a \ldots$, let ${D_0}^\sharp = {C_0}^\sharp$ and ${D_{i+1}}^\sharp = {D_i}^\sharp \nabla {C_{i+1}}^\sharp$. The sequence ${D_0}^\sharp \sqcup_a {D_1}^\sharp \sqcup_a \ldots$ stabilizes.
  - Convergence guaranteed when using ∇ instead of $\sqcup_a$.

Widening for the interval domain

- A widening operator ∇ for the interval domain:
  - $[a, b] \nabla \bot = \bot \nabla [a, b] = [a, b]$
  - $[a, b] \nabla [c, d] = [l, r]$ with
    - $l = a$ if $a \leq c$ and $l = -\infty$ otherwise
    - $r = b$ if $b \geq d$ and $r = \infty$ otherwise
- $[3, 10] \nabla [2, 9]$
- ...

Fixpoint computation: widening (1)



$[0,0], [0,1], [0,2], [0,3]....$
would take long time to converge.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:⊤
L1. x:= 0
//x:[0,0]
L2. x:= x + 1
//x: [1,1]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

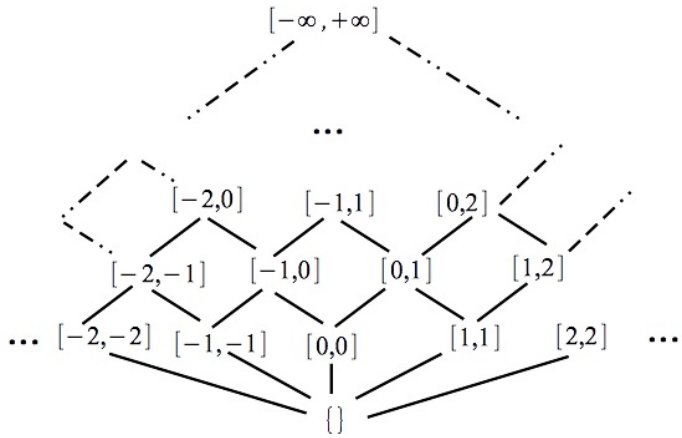
⟶

```
//x:⊤∇⊥
L1. x:= 0
//x:([0,0] ∇ [0,0])
//      ∇([1,1] ⊓ [−oo,99]
L2. x:= x + 1
//x:[1,1] ∇ [1,1]
L3. if x < 100 goto L2
//x:⊥∇([1,1] ⊓ [100,+oo])
L4. assert x >= 100
//x:⊥
L5. end
```

⟶

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x:[1,1]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

Fixpoint computation: widening (2)



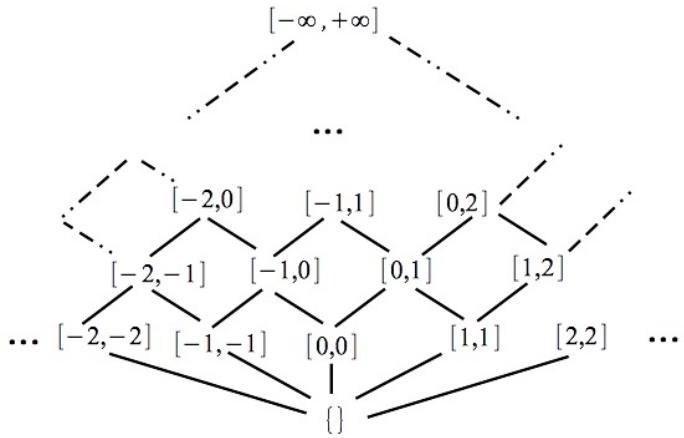
$[0,0], [0,1], [0,2], [0,3]....$
would take long time to converge.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x: [1,1]
L3. if x < 100 goto L2 ⟶
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

```
//x:⊤∇⊥
L1. x:= 0
//x:([0,+oo] ∇ [0,0])
//      ∇([1,1] ⊓ [−oo,99]
L2. x:= x + 1
//x:[1,1] ∇ [1,+oo]
L3. if x < 100 goto L2 ⟶
//x:⊥∇([1,1] ⊓ [100,+oo])
L4. assert x >= 100
//x:⊥
L5. end
```

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

Fixpoint computation: widening (3)



$[0,0], [0,1], [0,2], [0,3]....$
would take long time to converge.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x: [1,+oo]
L3. if x < 100 goto L2
//x:⊥
L4. assert x >= 100
//x:⊥
L5. end
```

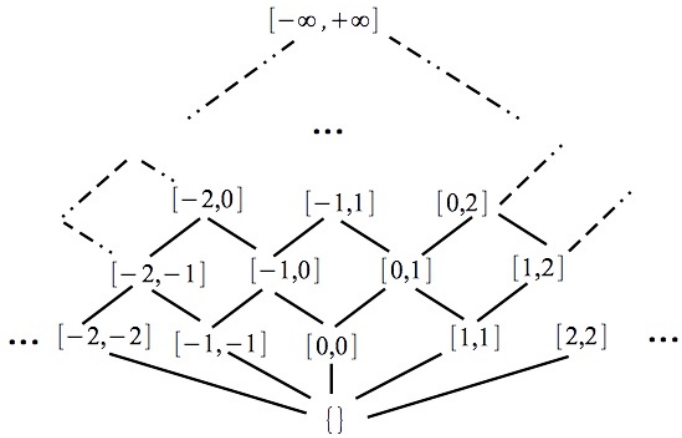
$\longrightarrow$

```
//x:⊤∇⊥
L1. x:= 0
//x:([0,+oo] ∇ [0,0])
//      ∇([1,+oo] ⊓ [−oo,99]
L2. x:= x + 1
//x:[1,+oo] ∇ [1,+oo]
L3. if x < 100 goto L2
//x:⊥∇([1,+oo] ⊓ [100,+oo])
L4. assert x >= 100
//x:⊥∇(⊥ ⊓ [100,+oo])
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x >= 100
//x:⊥
L5. end
```

Fixpoint computation: widening (4)



$[0,0], [0,1], [0,2], [0,3]....$
would take long time to converge.
For this use some widening operator ∇.
Intuitively, an acceleration that ensures termination

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:⊥
L5. end
```

⟶

```
//x:⊤
L1. x:= 0
//x:([0,+oo]∇[0,0])
//      ∇[1,+oo]
L2. x:= x + 1
//x:[1,+oo]∇[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]∇([1,+oo]⊓[100,+oo])
L4. assert x <= 100
//x:⊥∇([100,+oo]⊓[−oo,100])
L5. end
```

⟶

```
//x:⊤
L1. x:= 0
//x:[0,oo]
L2. x:= x + 1
//x:[1,oo]
L3. if x < 100 goto L2
//x:[100,oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

Widening can overapproximate too much: narrowing

- After widening, having obtained a too coarse over-approximation of the lfp, use a narrowing operator Δ to regain some precision.
- Conditions on a narrowing operator Δ
  - for any abstract elements $A^\sharp \sqsubseteq_a B^\sharp$, $(A^\sharp \sqsubseteq_a (A^\sharp \Delta B^\sharp) \sqsubseteq_a B^\sharp)$
  - For any ${C_0}^\sharp \sqsupseteq_a {C_1}^\sharp \sqsupseteq_a \ldots$, let ${D_0}^\sharp = {C_0}^\sharp$ and ${D_{i+1}}^\sharp = {D_i}^\sharp \Delta {C_{i+1}}^\sharp$. The sequence ${D_0}^\sharp \sqcup_a {D_1}^\sharp \sqcup_a \ldots$ stabilizes.

Narrowing for the interval domain (1/4)

- A narrowing operator Δ for the interval domain:
  - $[a, b]\Delta\bot = [a, b]$
  - $[a, b]\Delta[c, d] = [l, r]$ with
    - $l = c$ if $a = -oo$ and $l = a$ otherwise
    - $r = d$ if $b = +oo$ and $r = b$ otherwise
- $[0, oo]\Delta[1, 99] = [0, 99]$

```
//x:⊤
L1. x:= 0
//x:[0,+oo]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:([0,+oo]Δ([0, 0] ⊔ [−oo, 99])
L2. x:= x + 1
//x:[1,+oo]Δ[1, +oo]
L3. if x < 100 goto L2
//x:[100, +oo]Δ([1, +oo] ⊓ [100, +oo])
L4. assert x <= 100
//x:[100, 100]Δ([100, +oo] ⊓ [100, +oo])
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

Narrowing for the interval domain (2/4)

- A narrowing operator Δ for the interval domain:
  - $[a, b]\Delta\bot = [a, b]$
  - $[a, b]\Delta[c, d] = [l, r]$ with
    - $l = c$ if $a = -oo$ and $l = a$ otherwise
    - $r = d$ if $b = +oo$ and $r = b$ otherwise
- $[0, oo]\Delta[1, 99] = [0, 99]$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,+oo]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:([0,99]Δ([0,0] ⊔ [−oo,99])
L2. x:= x + 1
//x:[1,+oo]Δ[1,100]
L3. if x < 100 goto L2
//x:[100,+oo]Δ([1,+oo] ⊓ [100,+oo])
L4. assert x <= 100
//x:[100,100]Δ([100,+oo] ⊓ [100,+oo])
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,100]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

Narrowing for the interval domain (3/4)

- A narrowing operator Δ for the interval domain:
  - $[a, b]\Delta\bot = [a, b]$
  - $[a, b]\Delta[c, d] = [l, r]$ with
    - $l = c$ if $a = -oo$ and $l = a$ otherwise
    - $r = d$ if $b = +oo$ and $r = b$ otherwise
- $[0, oo]\Delta[1, 99] = [0, 99]$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,100]
L3. if x < 100 goto L2
//x:[100,+oo]
L4. assert x <= 100
//x:[100,100]
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:([0,99]Δ([0,0] ⊔ [−oo,99])
L2. x:= x + 1
//x:[1,100]Δ[1,100]
L3. if x < 100 goto L2
//x:[100,+oo]Δ([1,100] ⊓ [100,+oo])
L4. assert x <= 100
//x:[100,100]Δ([100,+oo] ⊓ [100,+oo])
L5. end
```

$\longrightarrow$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,100]
L3. if x < 100 goto L2
//x:[100,100]
L4. assert x <= 100
//x:[100,100]
L5. end
```

Narrowing for the interval domain (4/4)

- A narrowing operator Δ for the interval domain:
  - $[a, b]\Delta\bot = [a, b]$
  - $[a, b]\Delta[c, d] = [l, r]$ with
    - $l = c$ if $a = -oo$ and $l = a$ otherwise
    - $r = d$ if $b = +oo$ and $r = b$ otherwise
- $[0, oo]\Delta[1, 99] = [0, 99]$

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,100]
L3. if x < 100 goto L2
//x:[100,100]
L4. assert x <= 100
//x:[100,100]
L5. end
```

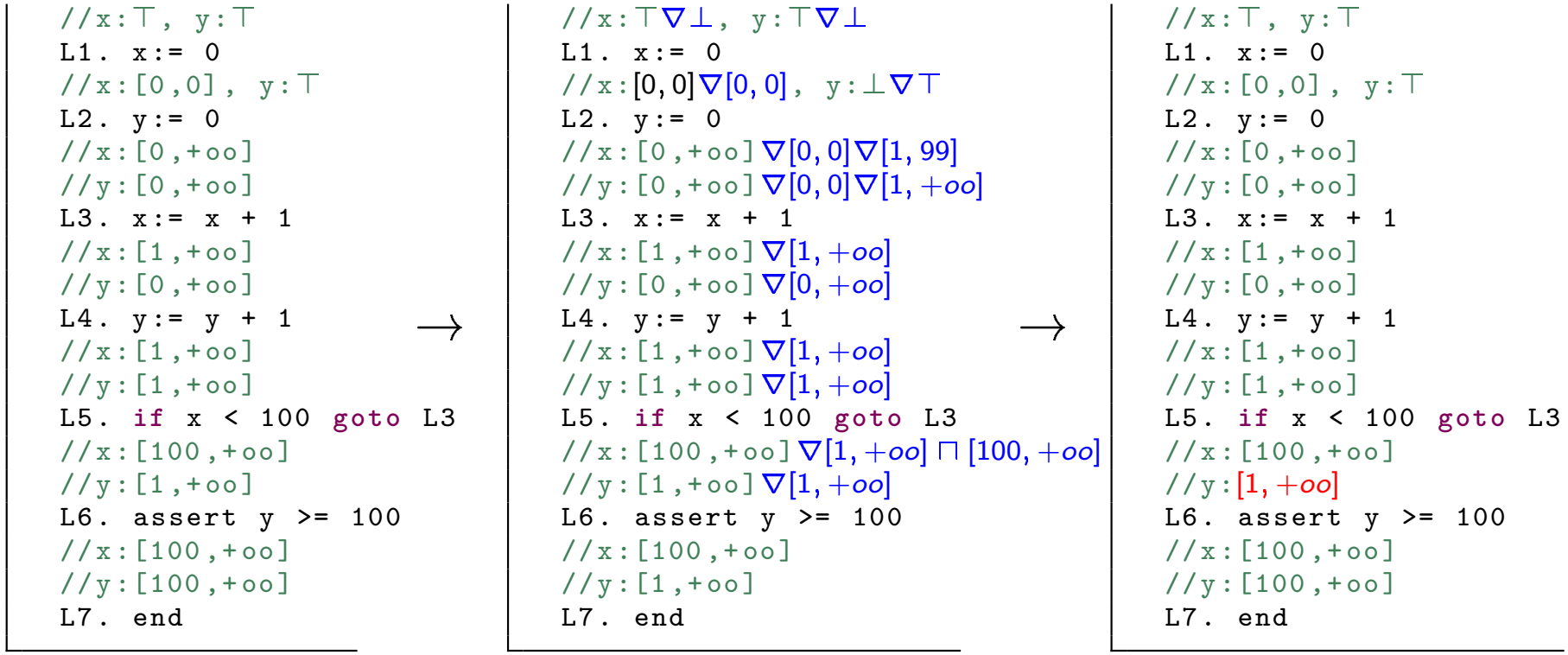
→

```
//x:⊤
L1. x:= 0
//x:([0,99]Δ([0,0] ⊔ [−oo,99])
L2. x:= x + 1
//x:[1,100]Δ[1,100]
L3. if x < 100 goto L2
//x:[100,100]Δ([1,100] ⊓ [100,+oo])
L4. assert x <= 100
//x:[100,100]Δ([100,100] ⊓ [100,+oo])
L5. end
```

→

```
//x:⊤
L1. x:= 0
//x:[0,99]
L2. x:= x + 1
//x:[1,100]
L3. if x < 100 goto L2
//x:[100,100]
L4. assert x <= 100
//x:[100,100]
L5. end
```

Need for relational domains



- Intervals do not capture relations between variables
- DBMs $x - y \leq k$
- Octagons $\pm x \pm y \leq k$
- Polyhedra $a_1 x_1 + \ldots + a_n x_n <= k$
- Shape analysis, arrays, lists, etc

Outline

Further readings


Apron numerical abstract domain library.
`http://apron.cri.ensmp.fr/library/`.
Accessed: 2021-01-14.


General information on the ppl.
`https://www.bugseng.com/products/ppl/documentation/user/ppl-user-1.2-html/index.html`.
Accessed: 2021-01-14.


B. Blanchet.
Introduction to abstract interpretation.
lecture script, 2002.


P. Cousot.
Abstract Interpretation Based Formal Methods and Future Challenges, pages 138–156.
Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.


P. Cousot and R. Cousot.
Systematic design of program analysis frameworks.
POPL '79, 1979.