Static Analysis: Overview, Syntactic Analysis and Abstract Interpretation

TDDC90: Software Security

Ahmed Rezine

IDA, Linköpings Universitet

Hösttermin 2024

Outline

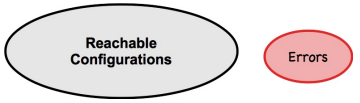
Static Program Analysis

Static Program Analysis analyses computer programs **statically**, i.e., without executing them (as opposed to *dynamic analysis* that does execute the programs wrt. some specific input):

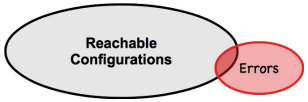
- No need to run programs, before deployment
- No need to restrict to a single input as for testing
- Useful in compiler optimization, program analysis, finding security vulnerabilities and verification
- Often performed on (models of) source code, sometimes on object code
- Usually highly automated though with the possibility of some user interaction
- From scalable bug hunting tools without guarantees to heavy weight verification frameworks for safety critical systems

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**



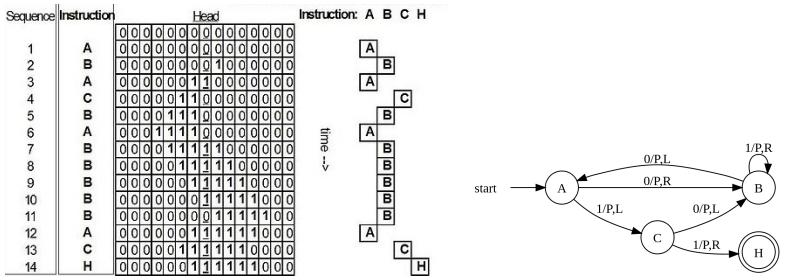
Safe Program



Unsafe Program

The general verification problem is "very difficult"

- Deciding whether all possible executions of a program are error-free is hard. If we could write a program that does it for arbitrary programs to be analyzed then we would always be able to answer whether a Turing machine halts.
- This problem is proven to be undecidable.



Problem is "very difficult": what to do?

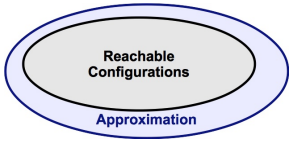
- Identify sub-problems on which one can decide: e.g. finite state machines, push-down automata, timed automata, Petri nets, well-structured transition systems.
- Proceed with approximations that will hopefully give some guarantees.

Verification problem and approximations

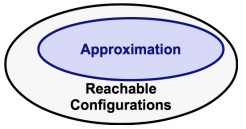
- An analysis procedure takes as input a program to be checked against a property. The procedure is an analysis algorithm if it is guaranteed to terminate.
- An analysis algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors (pessimistic analysis)
- An algorithm is **complete** in the case where each time it is given a program that is safe wrt. some errors, then it does report it to be safe wrt. those errors (optimistic analysis)
- In general, you have to give up on one of the three: termination, soundness or completeness.

Verification problem and approximations

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



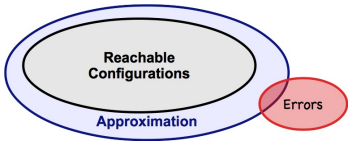
Over-approximation



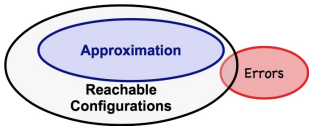
Under-approximation

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

These Two Lectures

These two lectures on static program analysis will briefly introduce different types of analysis:

- This lecture:
  - syntactic analysis: scalable but neither sound nor complete
  - abstract interpretation sound but not complete
- Next lecture:
  - symbolic executions: complete but not sound
  - inductive methods: may require heavy human interaction in proving the program correct
- These two lectures are only appetizers:
  - There is a deeper course with more tools and applications in the spring (TDDE34)
  - Possibilities of exjobbs with applications to verification.

Administrative Aspects:

- Lab sessions might not be enough and you might have to work outside these sessions
- You will need to write down your answers to each question on a draft.
- You will need to demonstrate (individually) your answers in a lab session on a computer to me or to Ulf.
- Once you get the green light, you can write your report in a pdf form and send it (in pairs) to the person you got the green light from.
- You will get questions in the final exam about these two lectures.

Outline

Automatic Unsound and Incomplete Analysis

- Tools such as the open source *Splint* or the heavier and commercial *Clockworck* and *Coverity* trade guarantees for scalability
- Not all reported errors are actual errors (false positives) and even if the program reports no errors there might still be uncovered errors (false negatives)
- A user needs therefore to carefully check each reported error, and to be aware that there might be more uncovered errors

Unsound and Incomplete analysis: Splint

- Some tools are augmented versions of grep and look for occurrences of memcpy, pointer dereferences ...
- The open source Splint tool checks C code for security vulnerabilities and programming errors.
- Splint does parse the source code and looks for certain patterns such as:
  - unused method parameters
  - loop tests that are not modified by the loop,
  - variables used before definitions,
  - null pointer dereference
  - overwriting allocated structures
  - and many more ...

Unsound and Incomplete analysis: Splint

```
...
 return *s; // warning about dereference of possibly null pointer s
...
if(s!=NULL)
  return *s; //does not give warnings because s was checked
```

```
int dumbfunc ()
{
  int i;
  if (i = 0) return 1;
  int j=i;
  while(i > 0){
    j--;
  }
  return 0;
}
```

Unsound and Incomplete analysis: Splint

- Still, the number of false positives remains very important, which may diminish the attention of the user since splint looks for “dangerous” patterns
- An important number of flags can be used to enable, inhibit or organize the kind of errors Splint should look for
- Splint gives the possibility to the user to annotate the source code in order to eliminate warnings
- Real errors can be made quite with annotations. In fact real errors will remain unnoticed with or without annotations

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:

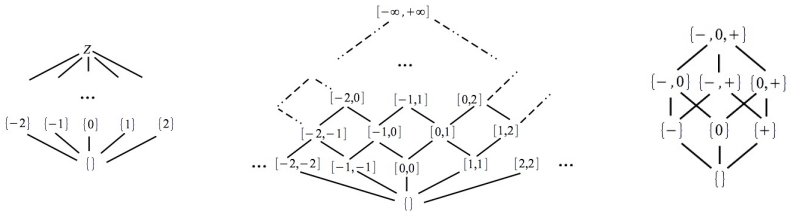
- Assume an "abstract domain" fixing the "form" of the tracked facts (e.g., $a \leq x \leq b$ or $a \leq x - y \leq b$ or ...)
- Define an abstract semantics that over-approximates the concrete semantics (e.g., abstract additions, substractions, comparisons, etc)
- Iterate computation with abstract semantics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

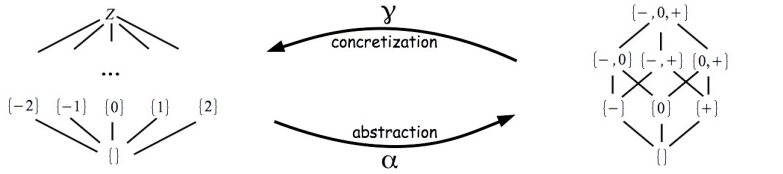
- A lattice is a poset $(Q, \sqsubseteq)$ where each pair p, q in Q has:
  - a greatest lower bound (aka. meet) $p \sqcap q$ wrt. $\sqsubseteq$, and
  - a least upper bound (aka. join) $p \sqcup q$ wrt. $\sqsubseteq$
- If S is a set, then $(\mathcal{P}(S), \subseteq, \bot, \top, \cup, \cap)$ is a complete 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

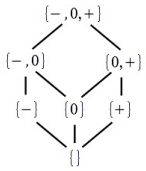
- An abstraction function: $\alpha(\{1, 10, 100, 103, 2021\}) = \{+\}$, $\alpha(\{-10, -3, 10\}) = \{-, +\}$, $\alpha(\{-3, 0\}) = \{-, 0\}$
- A concretization function: $\gamma(\{0\}) = \{0\}$, $\gamma(\{-, 0\}) = \{v | v \leq 0\}$ and $\gamma(\{+\}) = \{v | v > 0\}$



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
```

→

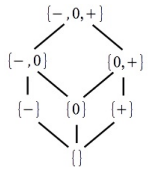
```
//x:⊤
//y:⊤
L1. x:= *
//x:
//y:
L2. y:= x
//x:
//y:
L3. if x > 0 goto L5
//x:
//y:
L4. end
//x:
//y:
L5. assert x >= 0
//x:
//y:
L6. end
```

→

```
//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
```

→

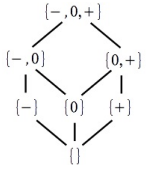
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊥⊔⊤
//y:⊥⊔⊤
L2. y:= x
//x:⊥⊔⊥
//y:⊥⊔⊥
L3. if x > 0 goto L5
//x:⊥⊔(⊥⊓{−,0})
//y:⊥⊔(⊥⊓⊤)
L4. end
//x:⊥⊔(⊥⊓{+})
//y:⊥⊔(⊥⊓⊤)
L5. assert x >= 0
//x:⊥⊔(⊥⊓{0,+})
//y:⊥⊔(⊥⊓⊤)
L6. end
```

→

```
//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
```

→

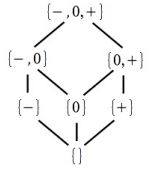
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y:= x
//x:⊥⊔⊤
//y:⊥⊔⊤
L3. if x > 0 goto L5
//x:⊥⊔(⊥ ⊓ {−,0})
//y:⊥⊔(⊥ ⊓ ⊤)
L4. end
//x:⊥⊔(⊥ ⊓ {+})
//y:⊥⊔(⊥ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔(⊥ ⊓ {0,+})
//y:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//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
```

→

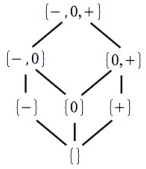
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y:= x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:⊥⊔(⊤ ⊓ {−,0})
//y:⊥⊔(⊤ ⊓ ⊤)
L4. end
//x:⊥⊔(⊤ ⊓ {+})
//y:⊥⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔(⊥ ⊓ {0,+})
//y:⊥⊔(⊥ ⊓ ⊤)
L6. end
```

→

```
//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
```

→

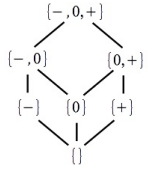
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y:= x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
//y:⊤⊔(⊤ ⊓ ⊤)
L4. end
//x:{+}⊔(⊤ ⊓ {+})
//y:⊤⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:⊥⊔({+} ⊓ {0,+})
//y:⊥⊔(⊤ ⊓ ⊤)
L6. end
```

→

```
//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
```

→

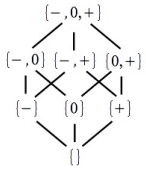
```
//x:⊤⊔⊥
//y:⊤⊔⊥
L1. x:= *
//x:⊤⊔⊤
//y:⊤⊔⊤
L2. y:= x
//x:⊤⊔⊤
//y:⊤⊔⊤
L3. if x > 0 goto L5
//x:{−,0}⊔(⊤ ⊓ {−,0})
//y:⊤⊔(⊤ ⊓ ⊤)
L4. end
//x:{+}⊔(⊤ ⊓ {+})
//y:⊤⊔(⊤ ⊓ ⊤)
L5. assert x >= 0
//x:{+}⊔({+} ⊓ {0,+})
//y:⊤⊔(⊤ ⊓ ⊤)
L6. end
```

→

```
//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: signs vs. intervals

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

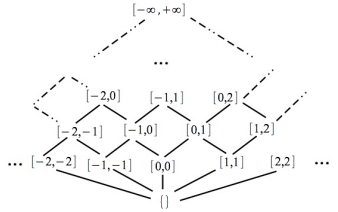



```
//x:⊤
L1. x:= 0
//x:{0}
L2. x:= x + 1
//x:{+}
L3. x:= x + 2
//x:{+}
L4. assert x >= 2
//x:{+}
L5. end
```

```
//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



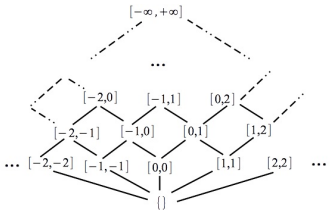
$[a,b] \sqsubseteq [c,d]$ iff $c \leq a$ and $b \leq d$
$[a,b] \sqcup [c,d]$ is $[inf\{a,c\}, sup\{b,d\}]$
$[a,b] \sqcap [c,d]$ is $[sup\{a,c\}, inf\{b,d\}]$

```
//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
```

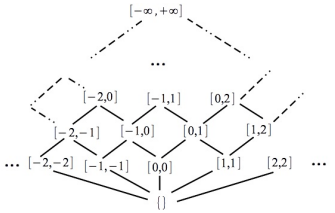
→

```
//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
```

→

```
//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:([0,25]∇[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
```

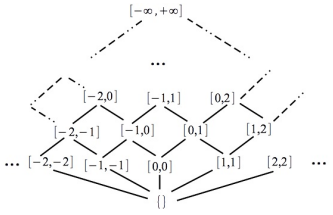
Fixpoint computation: widening

- 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
- $[3, 10]\nabla[2, 9]$
- ...

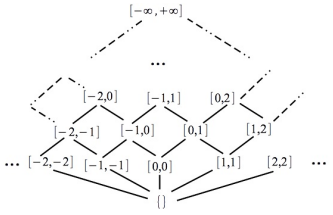
Fixpoint computation: widening (1)



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

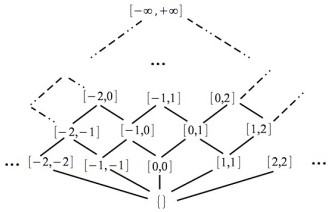
Fixpoint computation: widening (2)



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

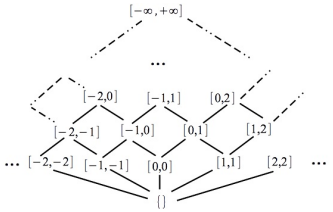
→

```
//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
```

→

```
//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
```

Need for relational domains

```
//x:⊤, y:⊤
L1. x:= 0
//x:[0,0], y:⊤
L2. y:= 0
//x:[0,+oo]
//y:[0,+oo]
L3. x:= x + 1
//x:[1,+oo]
//y:[0,+oo]
L4. y:= y + 1
//x:[1,+oo]
//y:[1,+oo]
L5. if x < 100 goto L3
//x:[100,+oo]
//y:[1,+oo]
L6. assert y >= 100
//x:[100,+oo]
//y:[100,+oo]
L7. end
```

→

```
//x:⊤∇⊥, y:⊤∇⊥
L1. x:= 0
//x:[0,0]∇[0,0], y:⊥∇⊤
L2. y:= 0
//x:[0,+oo]∇[0,0]∇[1,99]
//y:[0,+oo]∇[0,0]∇[1,+oo]
L3. x:= x + 1
//x:[1,+oo]∇[1,+oo]
//y:[0,+oo]∇[0,+oo]
L4. y:= y + 1
//x:[1,+oo]∇[1,+oo]
//y:[1,+oo]∇[1,+oo]
L5. if x < 100 goto L3
//x:[100,+oo]∇[1,+oo] ⊓ [100,+oo]
//y:[1,+oo]∇[1,+oo]
L6. assert y >= 100
//x:[100,+oo]
//y:[1,+oo]
L7. end
```

→

```
//x:⊤, y:⊤
L1. x:= 0
//x:[0,0], y:⊤
L2. y:= 0
//x:[0,+oo]
//y:[0,+oo]
L3. x:= x + 1
//x:[1,+oo]
//y:[0,+oo]
L4. y:= y + 1
//x:[1,+oo]
//y:[1,+oo]
L5. if x < 100 goto L3
//x:[100,+oo]
//y:[1,+oo]
L6. assert y >= 100
//x:[100,+oo]
//y:[100,+oo]
L7. end
```

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