Continuing Abstract Interpretation

We have seen:

1. How to compile abstract syntax trees into control-flow graphs
2. Lattices, as structures that describe abstractly sets of program states (facts)
3. (started) Transfer functions that describe how to update facts
   Reviewing and continuing with:
4. Iterative algorithm, examples, convergence

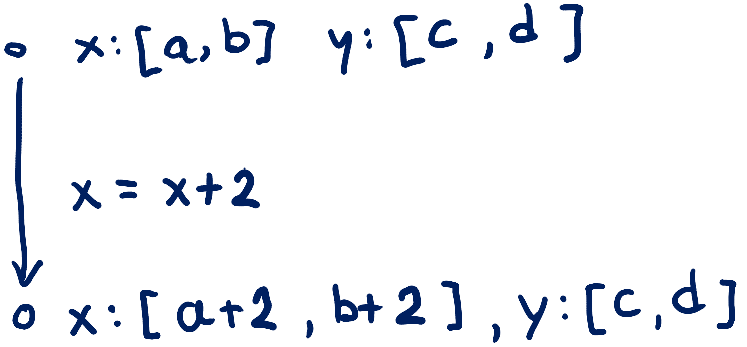
Defining Abstract Interpretation

Abstract Domain D (elements are data-flow **facts**), describing which information to compute, e.g.

- inferred **types** for each variable: x:C, y:D
- **interval** for each variable x:[a,b], y:[a',b']
- for each variable if is: **initialized**, **constant**, **live**

Transfer Functions, [[**st**]] for each statement **st**, how this statement affects the facts

- Example:



For now: domain of Intervals

- $D = \{\bot\} \cup \{ [a,b] \mid a \leq b, a,b \in \text{Int32}\}$

  Int64 = {-**MI**,...,-1,0,1,**MI**-1}, **MI** is e.g. 2^{63}

- Intervals [a,b] whose ends are machine integers a,b where a is less than or equal to b

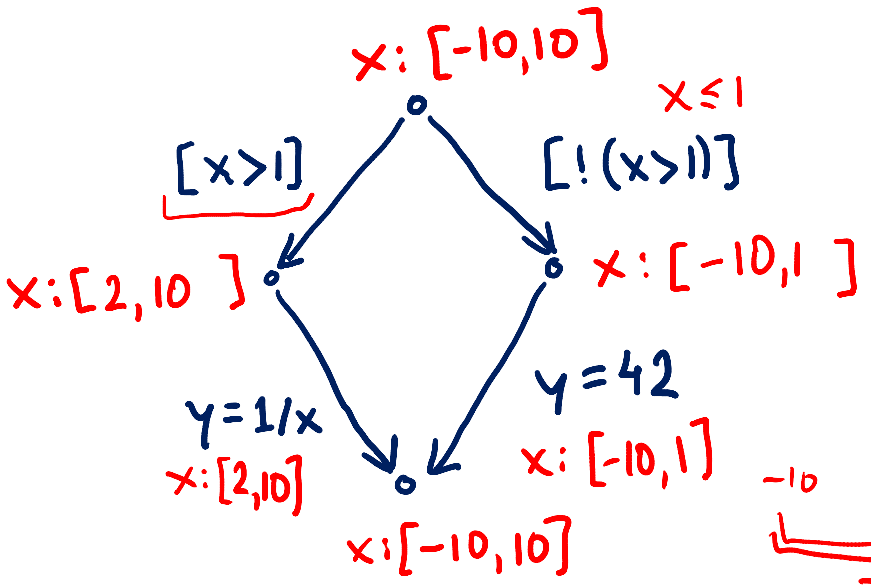
  $[a,b] = \{ x \mid a \leq x \leq b\}$

- When a is minimal int, b maximal int, we have the largest representable set, largest element of the lattice, we also denote this by T (top)
- Least element $\bot$, bottom represents empty set

Transfer Functions for **Tests**

Tests e.g. [x>1] come from translating if,while into CFG

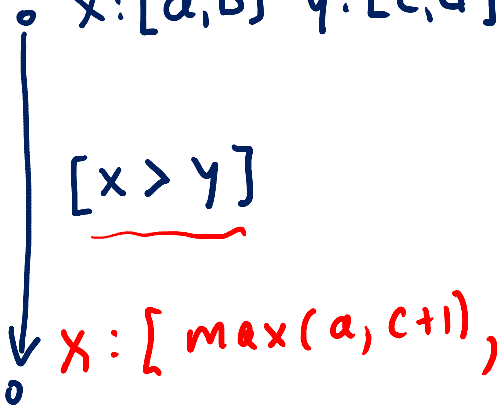
```
x : [-10,10]
if (x > 1) {
   x : [2,10]
   y = 1 / x
} else {
   x : [-10,1]
   y = 42
}
```



$[-10,-5] \sqcup [3,7] = [-10,7]$

$x \geq a$

$x \geq y+1 \geq c+1$



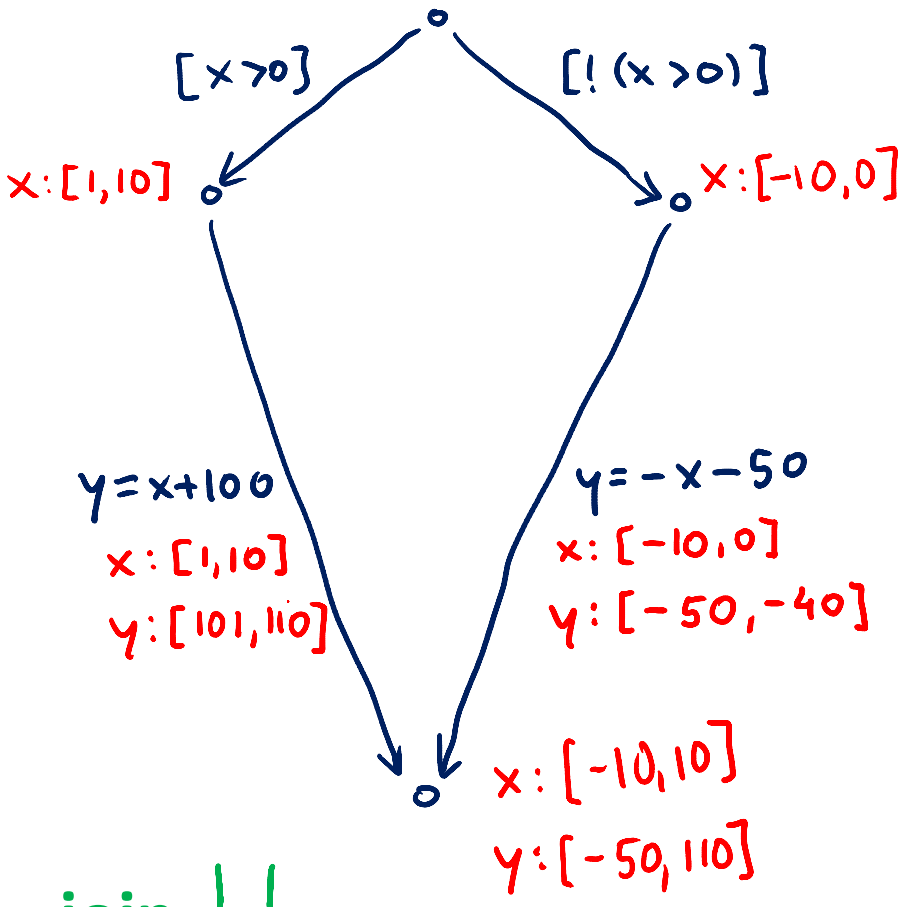
$x \leq b$

$y : [\ c\ ,\ \min(d, b-1)\ ]$

Joining Data-Flow Facts

x: [-10,10] y: [-1000,1000]

```
if (x > 0) {
   x:                    y:
   y = x + 100
   x:                    y:
} else {
   x:                    y:
   y = -x – 50
   x:                    y:
}
x:                       y:
```



join $\sqcup$

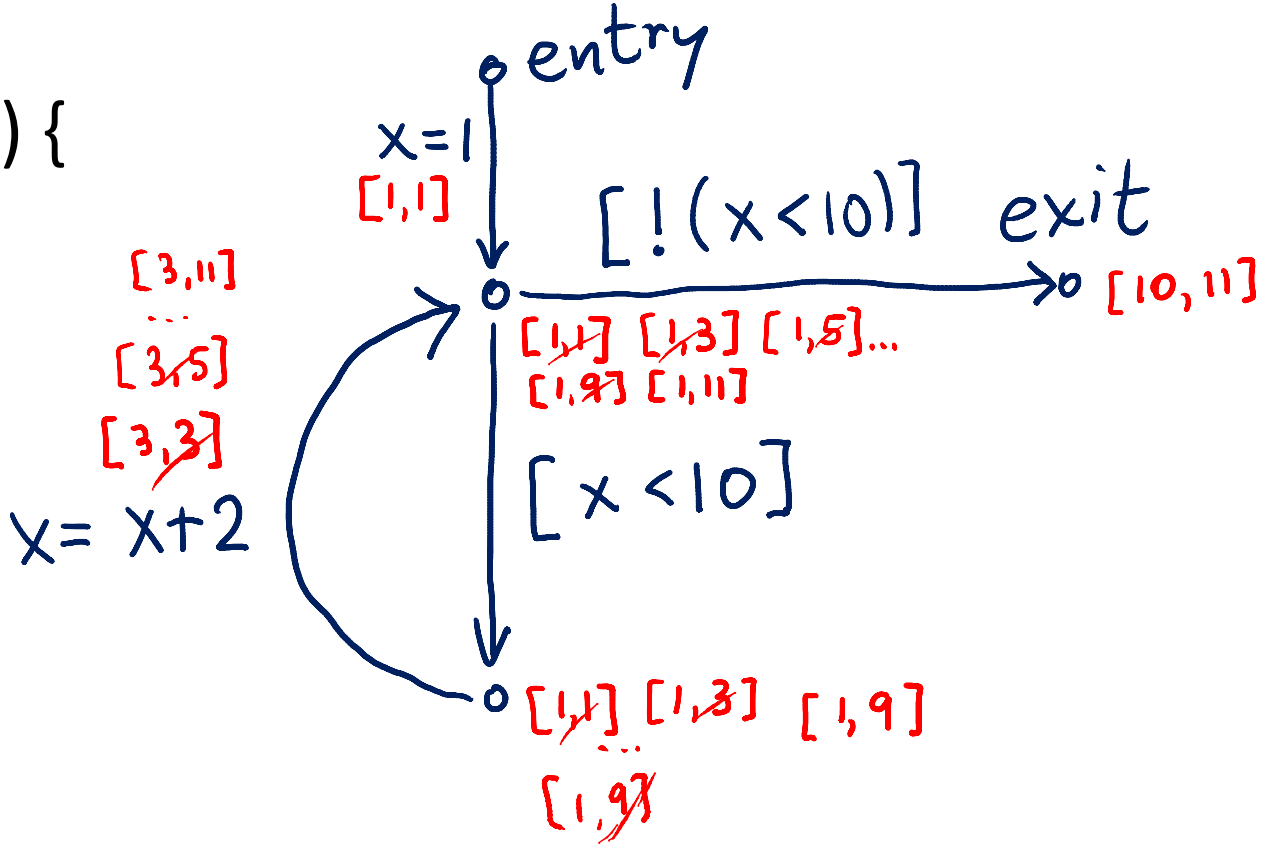
$[a,b] \sqcup [c,d] = [\min(a,c), \max(b,d)]$

Handling Loops: Iterate Until Stabilizes

$[1,1] \sqcup [3,3] = [1,3]$

$[1,1] \sqcup [3,5] = [1,5]$

```
x = 1
x ∈ [1,1]
while (x < 10) {
  x ∈ [1,9]
  x = x + 2
  x ∈ [3,11]
}
x ∈ [10,11]
```



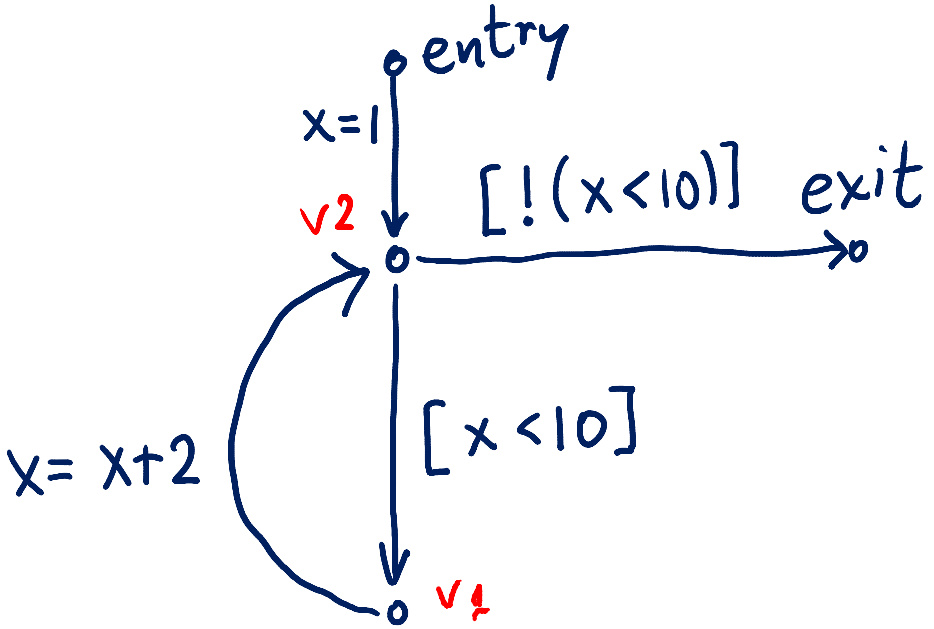
Analysis Algorithm

```
var facts : Map[Node,Domain] = Map.withDefault(empty)
facts(entry) = initialValues

while (there was change)
  pick edge (v1,statmt,v2) from CFG
          such that facts(v1) has changed
  facts(v2)=facts(v2) join transerFun(statmt, facts(v1))
}
```

⊔

Order does not matter for the end result, as long as we do not permanently neglect any edge whose source was changed.



```
var facts : Map[Node,Domain] = Map.withDefault(empty)
var worklist : Queue[Node] = empty
  def assign(v1:Node,d:Domain) = if (facts(v1)!=d) {
    facts(v1)=d
    for (stmt,v2) <- outEdges(v1) { worklist.add(v2) }
 }
assign(entry, initialValues)
while (!worklist.isEmpty) {
   var v2 = worklist.getAndRemoveFirst
   update = facts(v2)
   for (v1,stmt) <- inEdges(v2)
     { update = update join transferFun(facts(v1),stmt) }
   assign(v2, update)
}
```

Work List Version

Run range analysis, prove **error** is unreachable

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

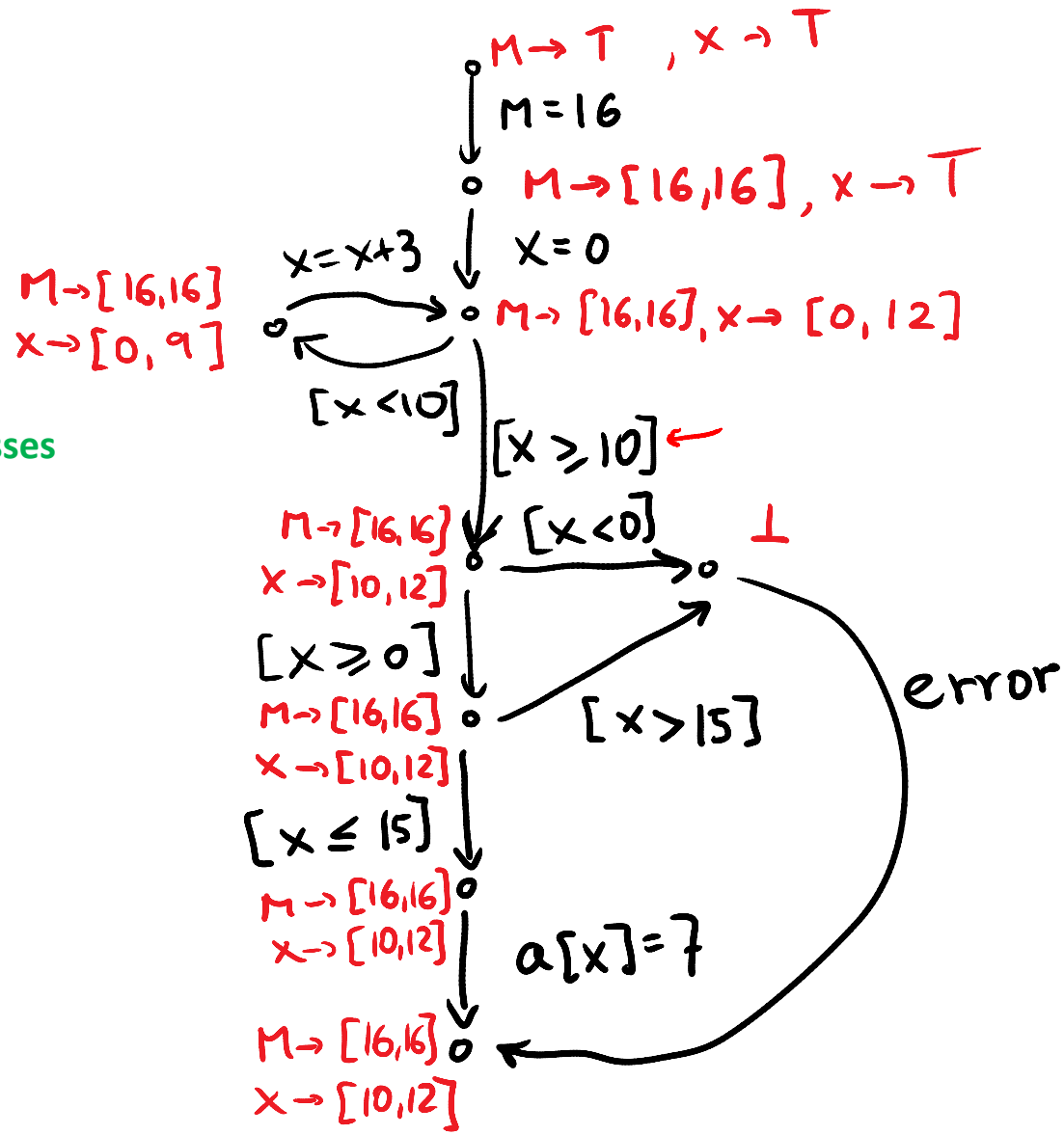
checks array accesses

M-1

Range analysis results

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

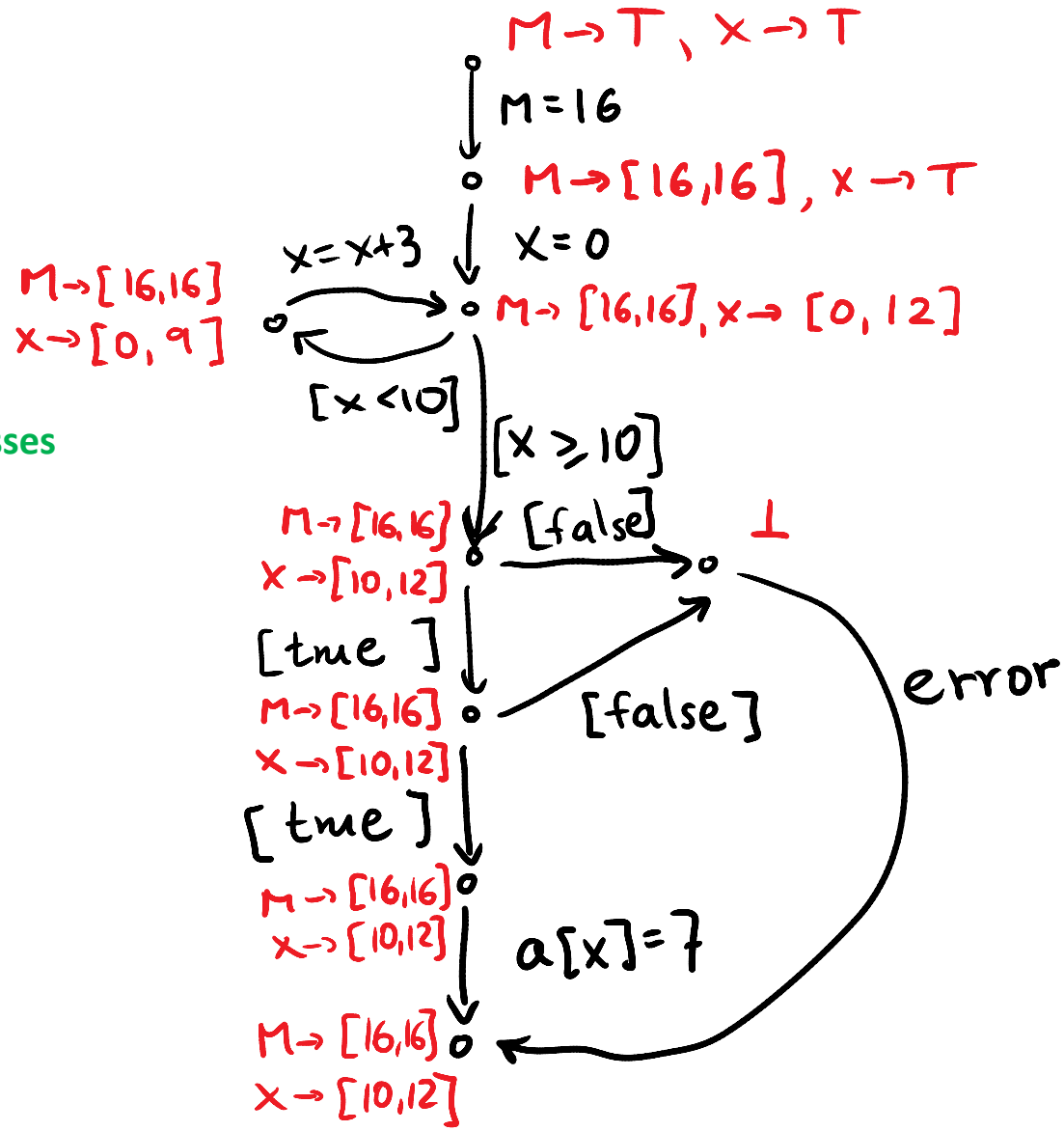
checks array accesses



Simplified Conditions

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

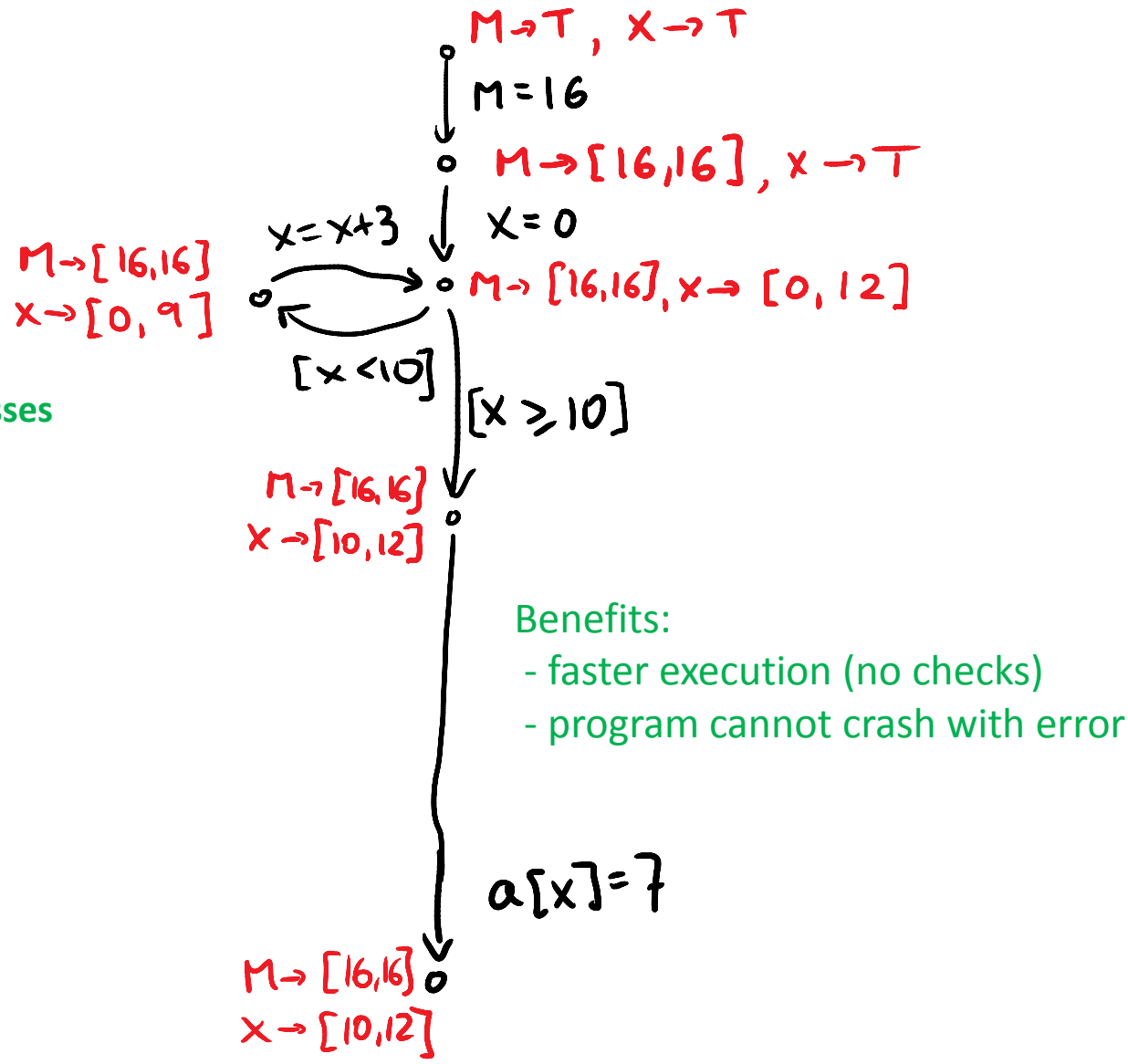
checks array accesses



Remove Trivial Edges, Unreachable Nodes

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

checks array accesses



Benefits:
- faster execution (no checks)
- program cannot crash with error

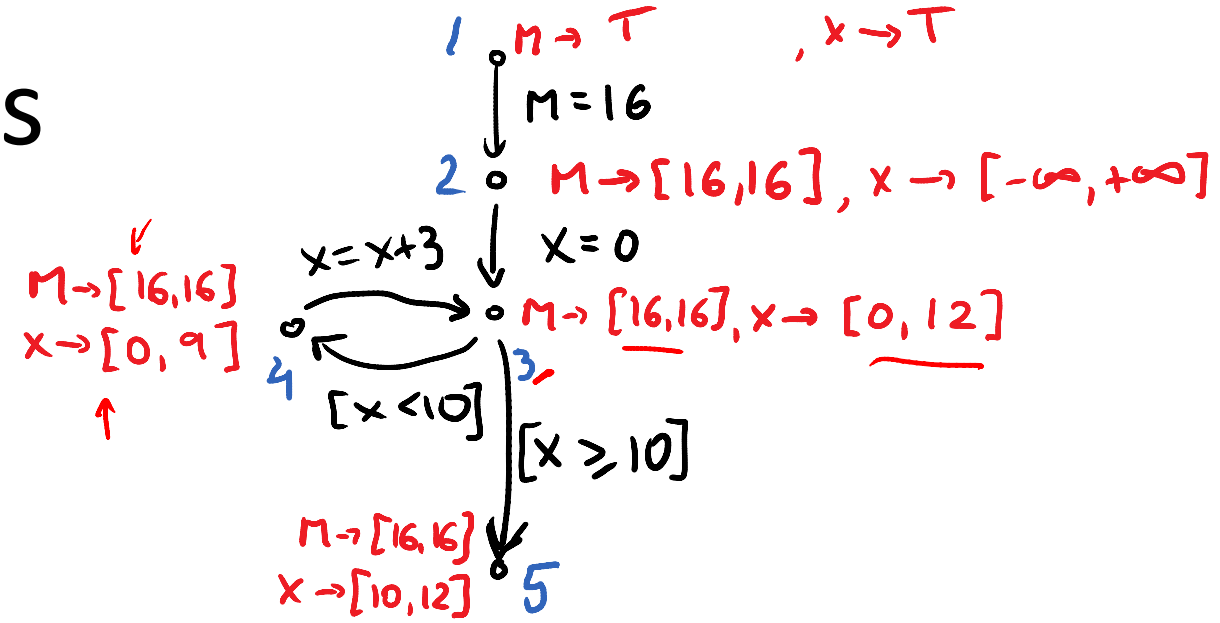
```
int a, b, step, i;
boolean c;
a = 0;
b = a + 10;
step = -1;
if (step > 0) {
  i = a;
} else {
  i = b;
}
c = true;
while (c) {
  process(i);
  i = i + step;
  if (step > 0) {
    c = (i < b);
  } else {
    c = (i > a);
  }
}
```

Apply Range Analysis and Simplify

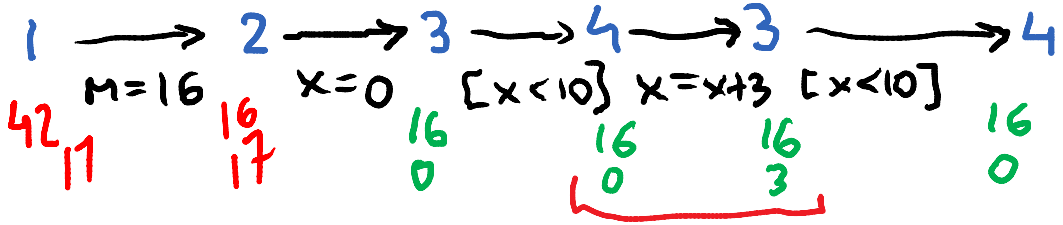
For booleans, use this lattice: D_b = { {}, {false}, {true}, {false,true} }
with ordering given by set subset relation.

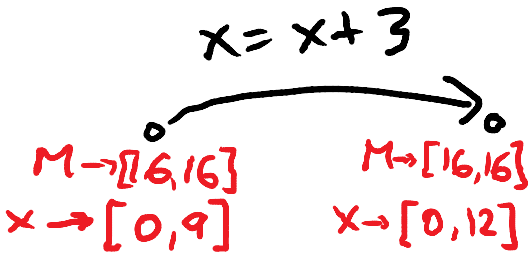
(left as exercise)

Correctness



Once the iterative loop stops, there were no changes. From there it follows:
All program states that flow along an edge are included in the states in the target node.
As long as this condition holds, it does not matter how we computed the states, the analysis results are correct.
Proof is by considering an execution sequence through CFG and showing by induction that each state in the sequence is contained in the intervals.



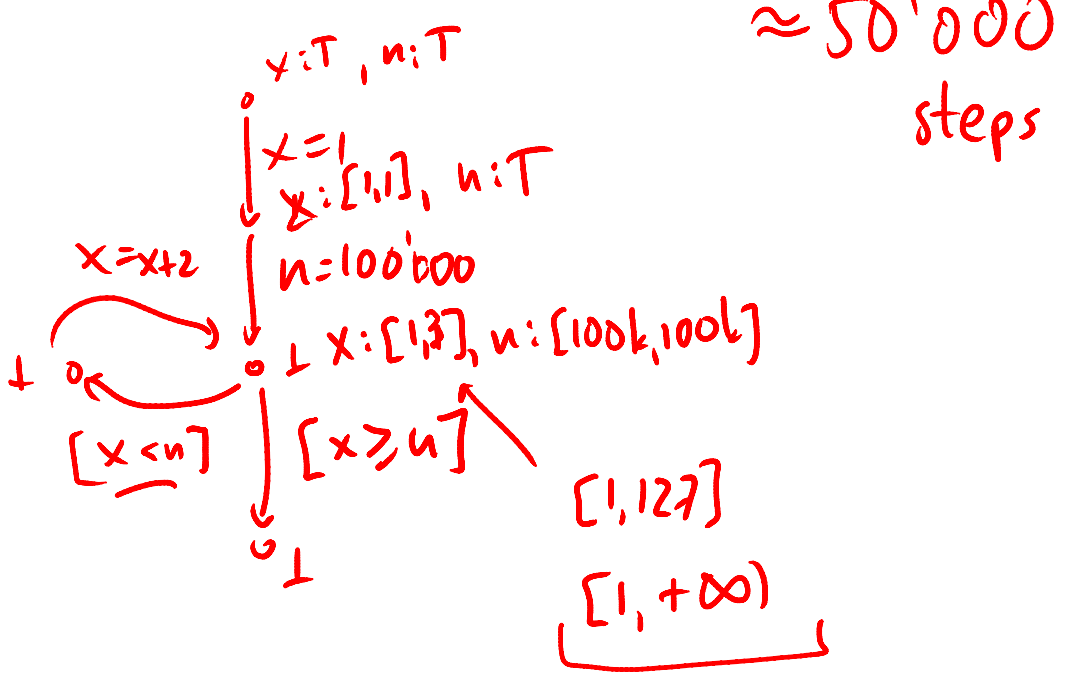


How Long Does Analysis Take?

Handling Loops: Iterate **Until Stabilizes**

How many steps until it stabilizes? Compute.

```
x = 1
n = 100000
while (x < n) {
  x = x + 2
}
```



≈ 50'000 steps

Handling Loops: Iterate **Until Stabilizes**

How many steps until it stabilizes?

x: [0, +∞) x ≥ 0

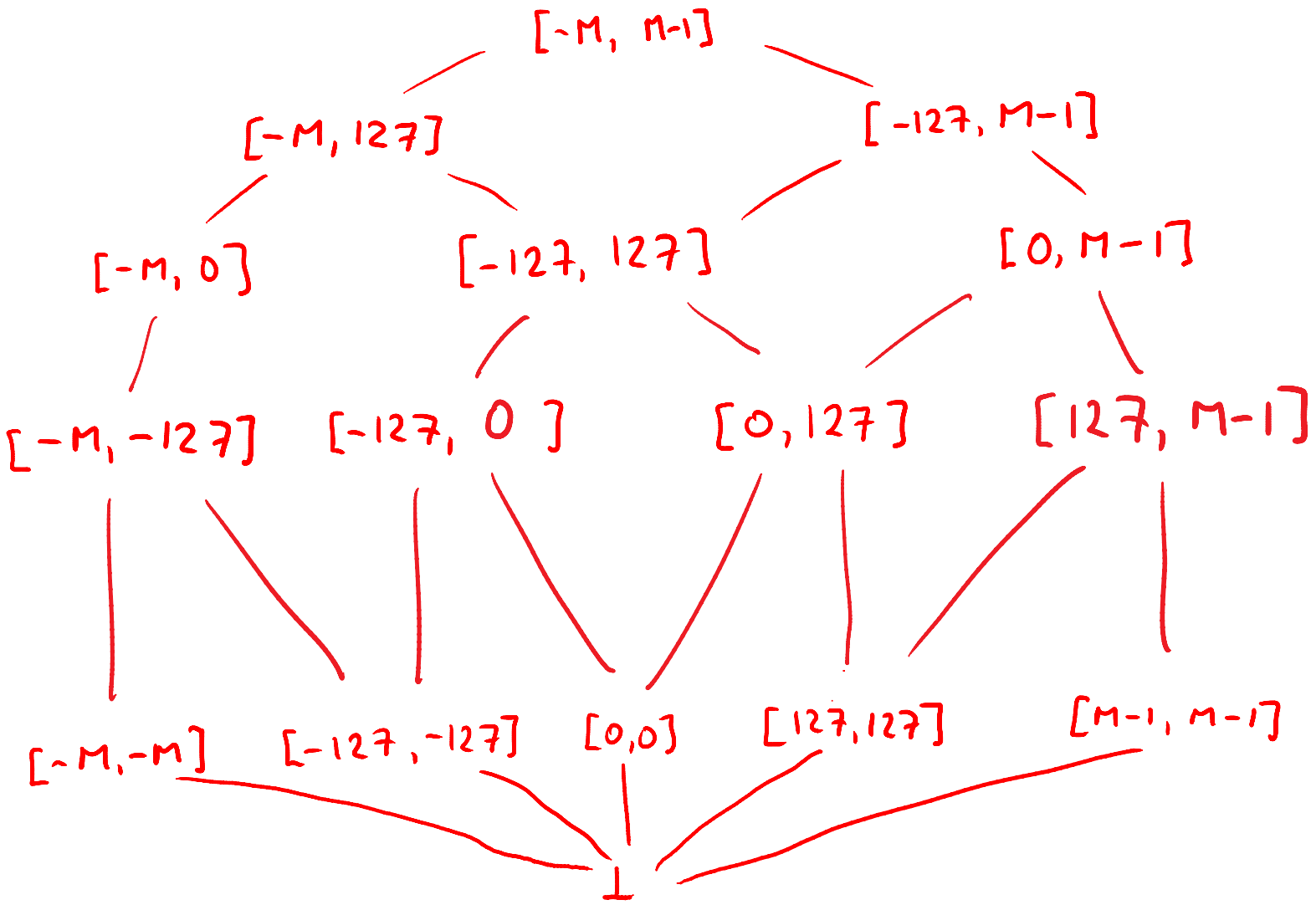
```
var x : BigInt = 1
var n : BigInt = readInput()
while (x < n) {
  x = x + 2
}
```

For unknown program inputs and unbounded domains it may be practically impossible to know how long it takes.

Solutions

- smaller domain, e.g. only certain intervals
  [a,b] where a,b in {-**MI**,-127,-1,0,1,127,**MI**-1}
- *widening* techniques (make it less precise on demand)

Smaller domain: intervals [a,b] where a,b∈{-**MI**,-127,0,127,**MI**-1} (**MI** denoted M)



Size of analysis domain

Interval analysis:

$D_1 = \{ [a,b] \mid a \le b,\ a,b \in \{-M,-127,-1,0,1,127,M-1\}\} \cup \{\bot\}$

Constant propagation:

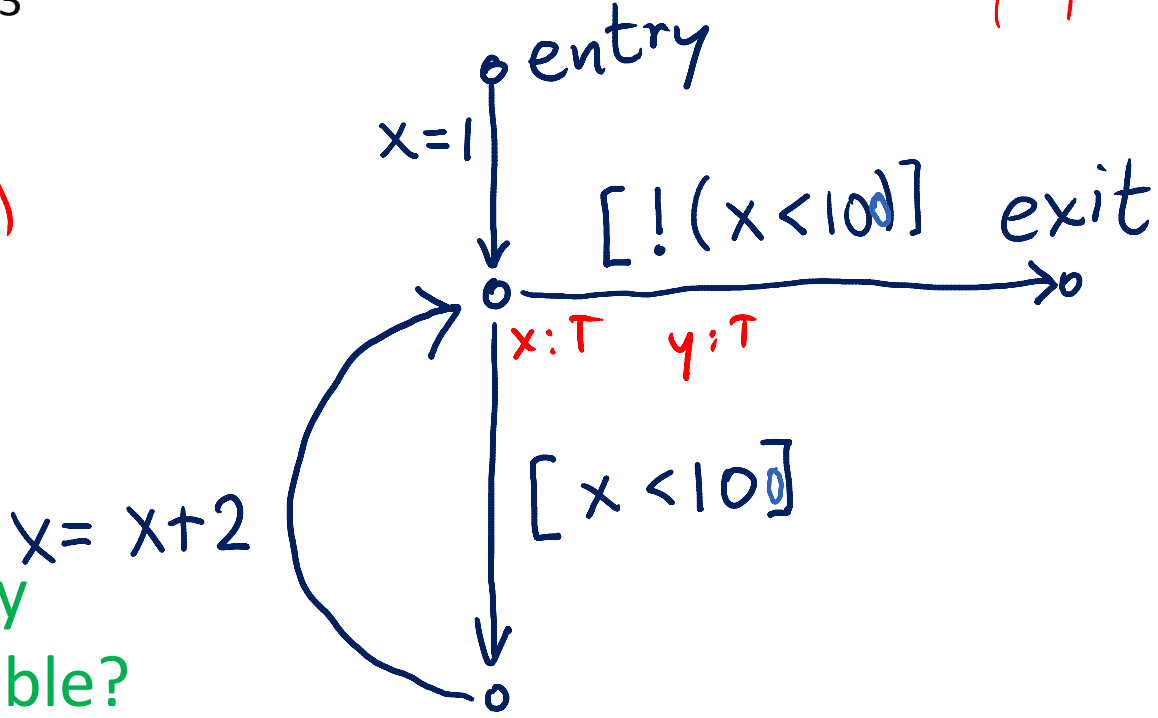
$D_2 = \{ [a,a] \mid a \in \{-M,-(M-1),\ldots,-2,-1,0,1,2,3,\ldots,M-1\}\} \cup \{\bot, T\}$

suppose M is 2^{63}

$|D_1| = 1 + (7+6+5+\cdots+1)$

$|D_2| = 1 + 2^{64} + 1$

How many steps until it stabilizes, for any program with one variable?



How many steps does the analysis take to finish (converge)?

Interval analysis:

$D_1 = \{ [a,b] \mid a \le b,\ a,b \in \{-M,-127,-1,0,1,127,M-1\}\} \cup \{\bot\}$

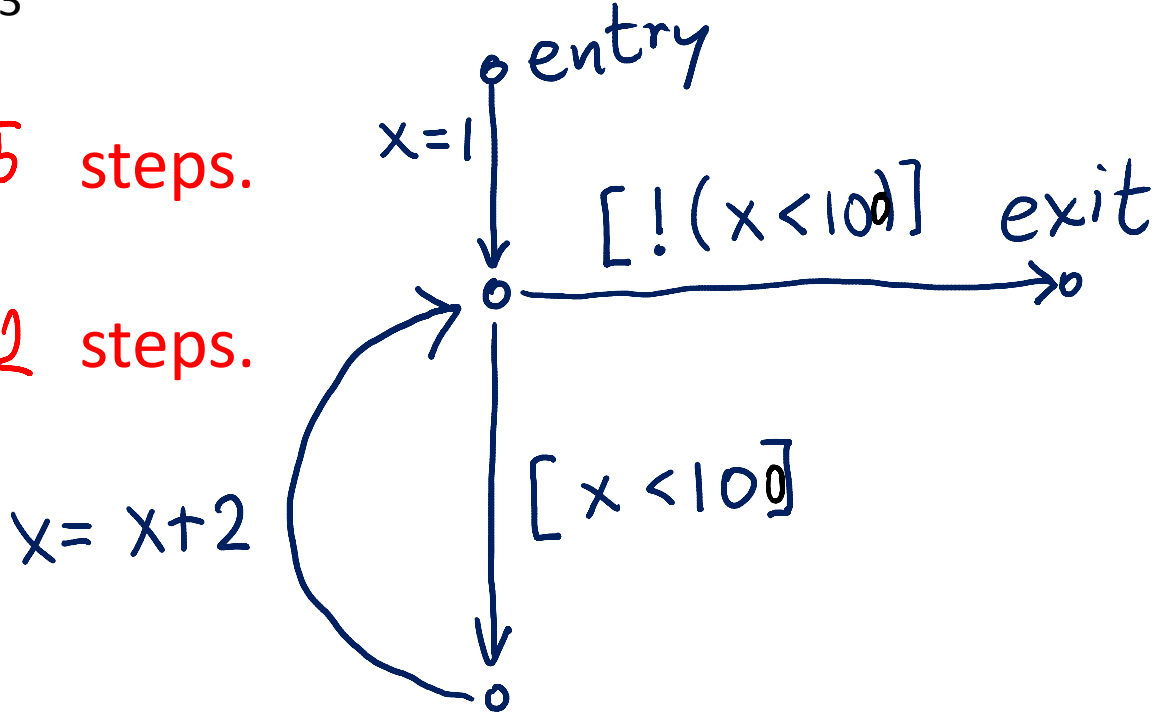
Constant propagation:

$D_2 = \{ [a,a] \mid a \in \{-M,-(M-1),\ldots,-2,-1,0,1,2,3,\ldots,M-1\}\} \cup \{\bot,T\}$

suppose M is 2^{63}

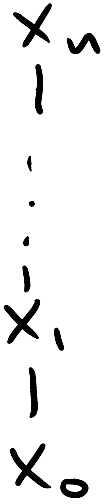
With D_1 takes at most 5 steps.

With D_2 takes at most 2 steps.



Chain of length n

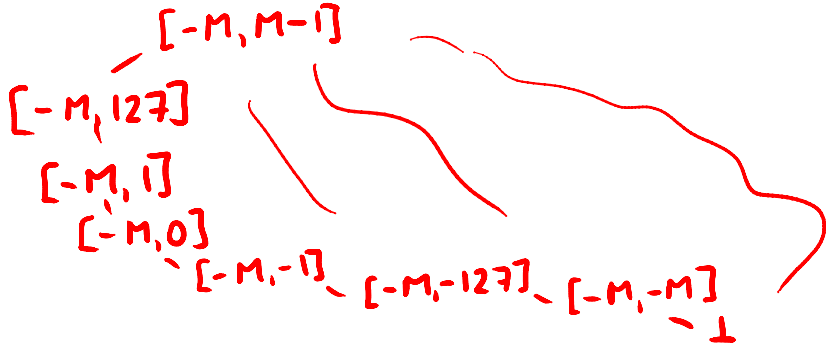
- A set of elements $x_0, x_1, \ldots, x_n$ in D that are linearly ordered, that is $x_0 \leq x_1 \leq \ldots \leq x_n$
- A lattice can have many chains. Its **height** is the maximum n for all the chains, if finite
- If there is no upper bound on lengths of chains, we say lattice has infinite height
- A monotonic sequence of distinct elements has length at most equal to lattice height



Termination Given by Length of Chains

Interval analysis:

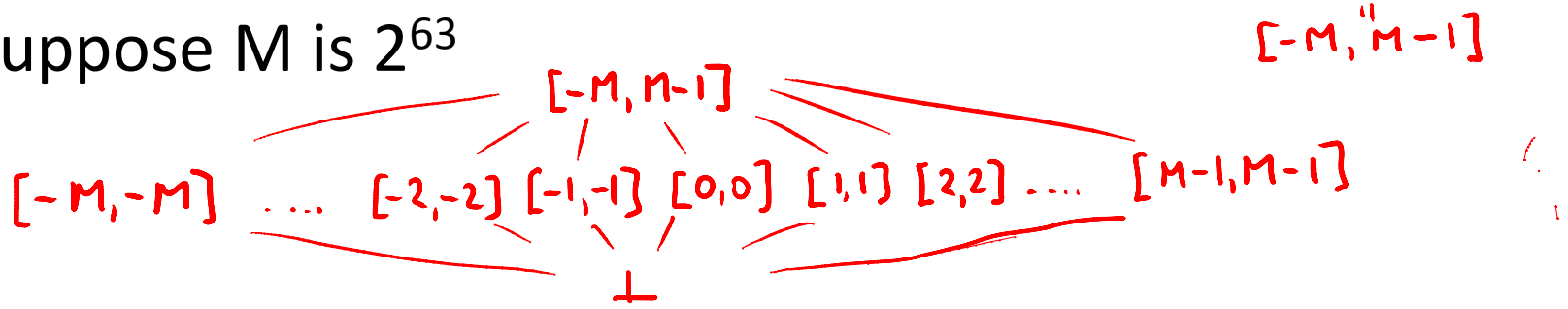
$D_1 = \{ [a,b] \mid a \leq b,\ a,b \in \{-M,-127,-1,0,1,127,M-1\} \} \cup \{\bot\}$



Constant propagation:

$D_2 = \{ [a,a] \mid a \in \{-M,\ldots,-2,-1,0,1,2,3,\ldots,M-1\} \} \cup \{\bot,T\}$

suppose M is 2^{63}



Product Lattice for All Variables

- If we have N variables, we keep one element for each of them
- This is like N-tuple of variables
- Resulting lattice is product of N lattices for individual variables
- Size is $|D|^N$
- The height is only N times the height of D

x: ⊥ y: ⊥
x: [0,0] y: ⊥
x: ⊤ y: ⊥
x: ⊤, y: [1,1]
x: ⊤, y: ⊤

$(2+2^{64})^{20}$

Max height: h(D) · |V| · |nodes| 2 · 20

$(\bot, \bot) \in D^2$

D^N

Relational Analysis

Suppose we keep track of interval for each var

...

```
// x:[0,10], y:[0,10]
if (x > y) {
  if (y > 3) {
     t = 100/(x - 4)
  }
}
```

We would not be able to prove that x-4 > 0.

Relational analysis would remember the constraint x<y to make such reasoning possible.

Instead of tracking bounds on x,y, it tracks also bounds on x-y and x+y.