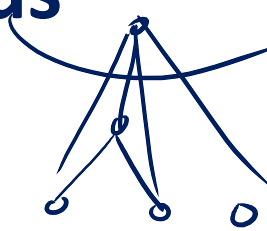
Abstract Interpretation

Lattice

Partial order: binary relation $\leq$ (subset of some D^2) which is

- reflexive: $x \leq x$
- anti-symmetric: $x \leq y \wedge y \leq x \rightarrow x = y$
- transitive: $x \leq y \wedge y \leq z \rightarrow x \leq z$

Lattice is a partial order in which every **two-element** set has **least among its upper bounds** and **greatest among its lower bounds**



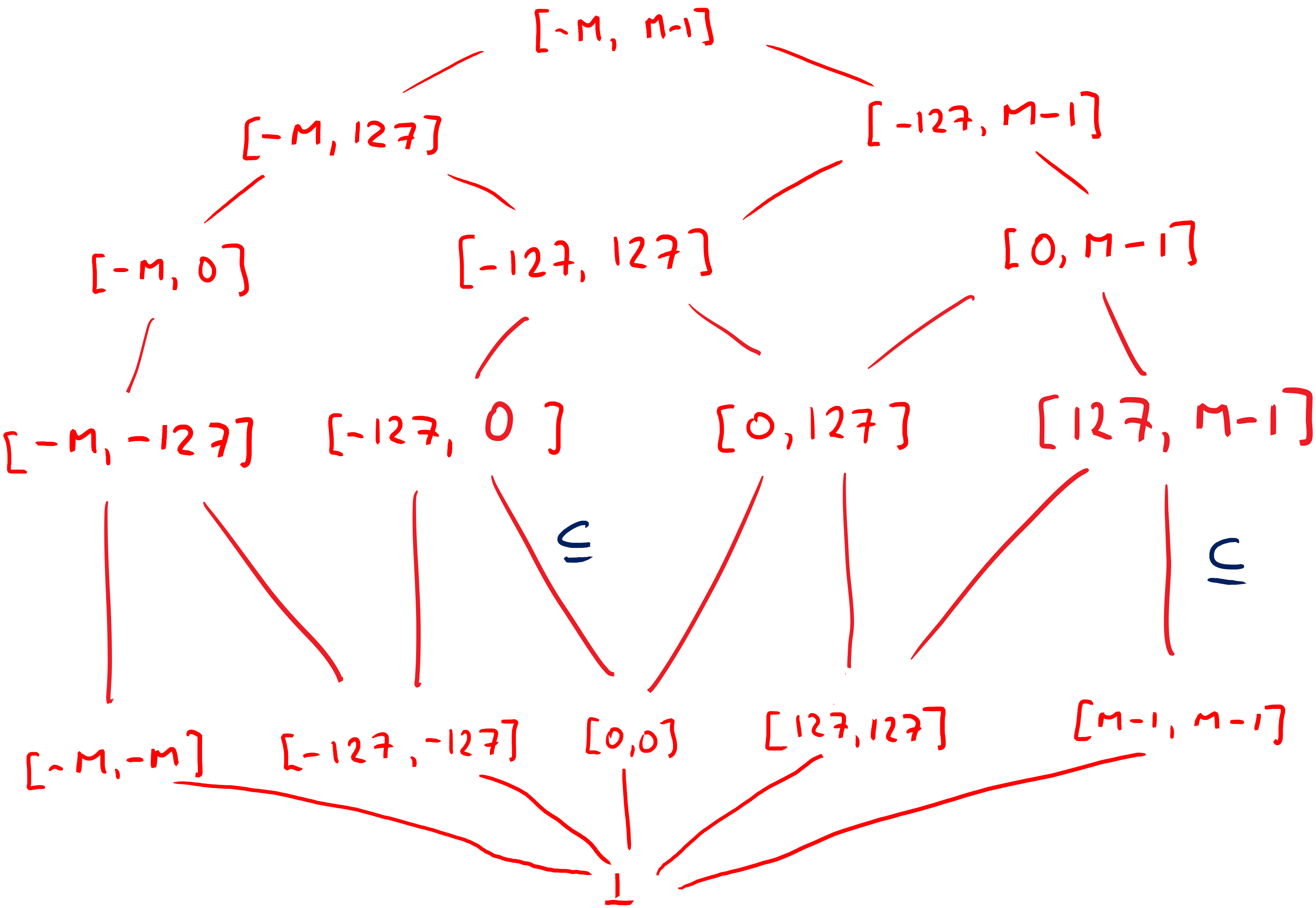
- Lemma: if $(D, \leq)$ is lattice and D is finite, then lub and glb exist for every finite set

$\sqcap$ $\sqcup$ $\sqcup \{a, b, c\}$

Graphs and Partial Orders

- If the domain is finite, then partial order can be represented by directed graphs
  - if $x \leq y$ then draw edge from x to y
- For partial order, no need to draw $x \leq z$ if $x \leq y$ and $y \leq z$. So we only draw non-transitive edges
- Also, because always $x \leq x$, we do not draw those self loops
- Note that the resulting graph is acyclic: if we had a cycle, the elements must to be equal

Domain of Intervals [a,b] where $a,b \in \{-M,-127,0,127,M-1\}$



Defining Abstract Interpretation

Abstract Domain D describing which information to compute – this is often a lattice

- inferred types for each variable: x:T1, y:T2
- interval for each variable   x:[a,b], y:[a',b']

Transfer Functions, [[**st**]] for each statement **st**, how this statement affects the facts $D \to D$

- Example:

$x:[a,b] \quad y:[c,d]$

$x = x+2$

$x:[a+2, b+2], \; y:[c,d]$

$$[\![x = x+2]\!](x:[a,b], \ldots) = (x:[a+2, b+2], \ldots)$$

For now, we consider arbitrary integer bounds for intervals

- Thus, we work with BigInt-s
- Often we must analyze machine integers
  - need to correctly represent (and/or warn about) overflows and underflows
  - fundamentally same approach as for unbounded integers
- For efficiency, many analysis do not consider arbitrary intervals, but only a subset of them W
- We consider as the domain
  - empty set (denoted $\perp$, pronounced "bottom")
  - all intervals [a,b] where a,b are integers and $a \leq b$, or where we allow $a= -\infty$ and/or $b = \infty$
  - set of all integers $[-\infty ,\infty]$ is denoted T , pronounced "top"

Find Transfer Function: Plus

Suppose we have only two integer variables: x,y

$x: [a,b]$ $\quad$ $y: [c,d]$

$x = x+y$

$x: [a',b']$ $\quad$ $y: [c',d']$

If $a \le x \le b$ $\qquad$ $c \le y \le d$

and we execute **x= x+y**

then $x' = x+y$

$y' = y$

so

$\le x' \le$

$\le y' \le$

So we can let

a'= a+c $\qquad$ b' = b+d

c'=c $\qquad$ d' = d

Find Transfer Function: Minus

Suppose we have only two integer variables: x,y

$x : [a, b] \quad y : [c, d]$

$y = x - y$

$x : [a', b'] \quad y : [c', d']$

If

and we execute **y= x-y**

then

So we can let

a’= a b’ = b

c’= a - d d’ = b - c

Transfer Functions for **Tests**

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

x : [-10,10]

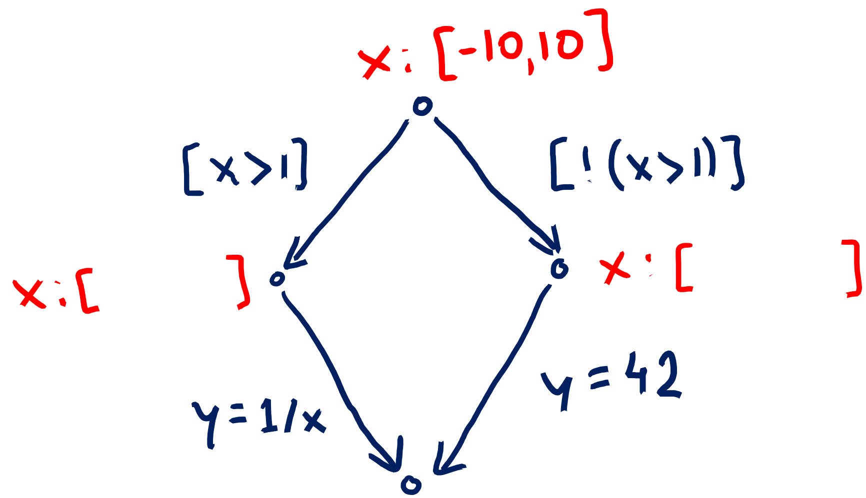
```
if (x > 1) {
```

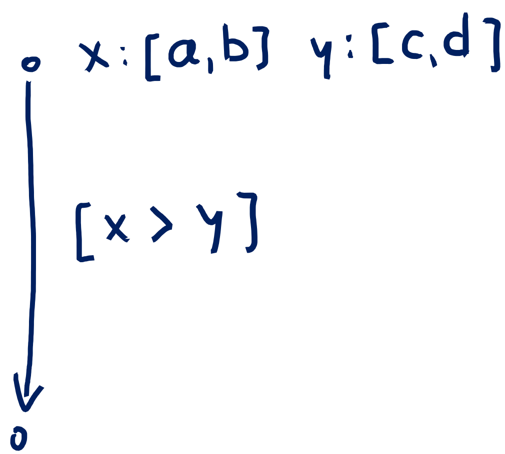
x :

```
  y = 1 / x
} else {
```

x :

```
  y = 42
}
```

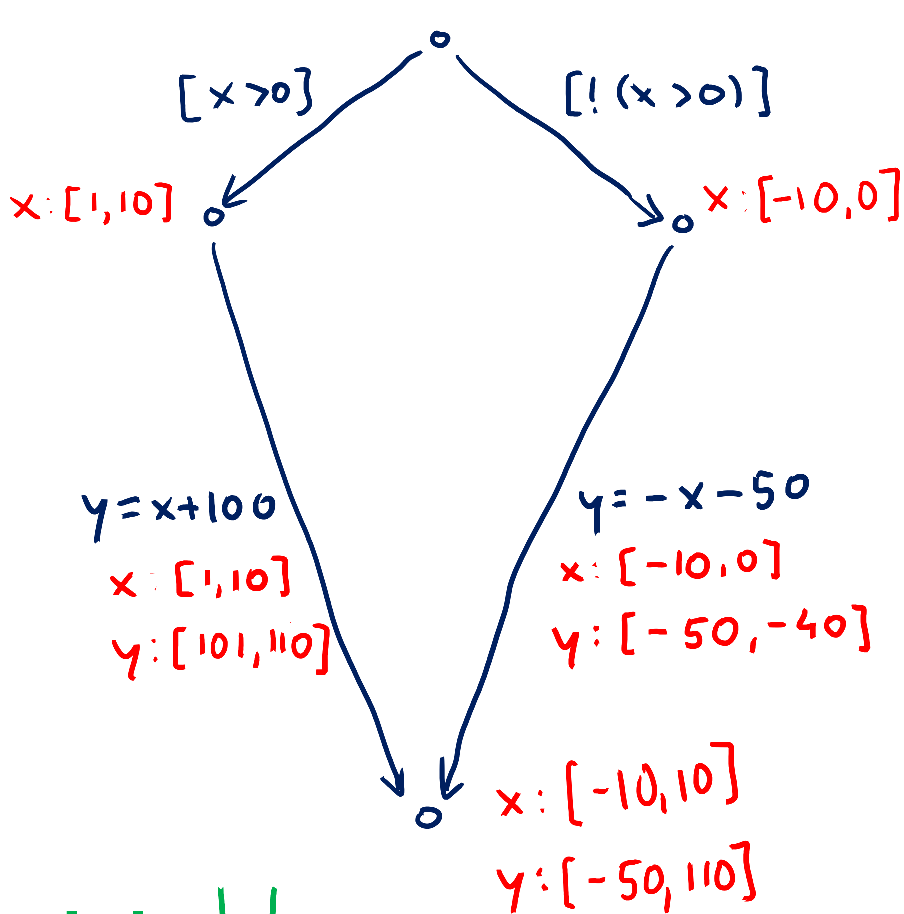




Joining Data-Flow Facts

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 ⊔

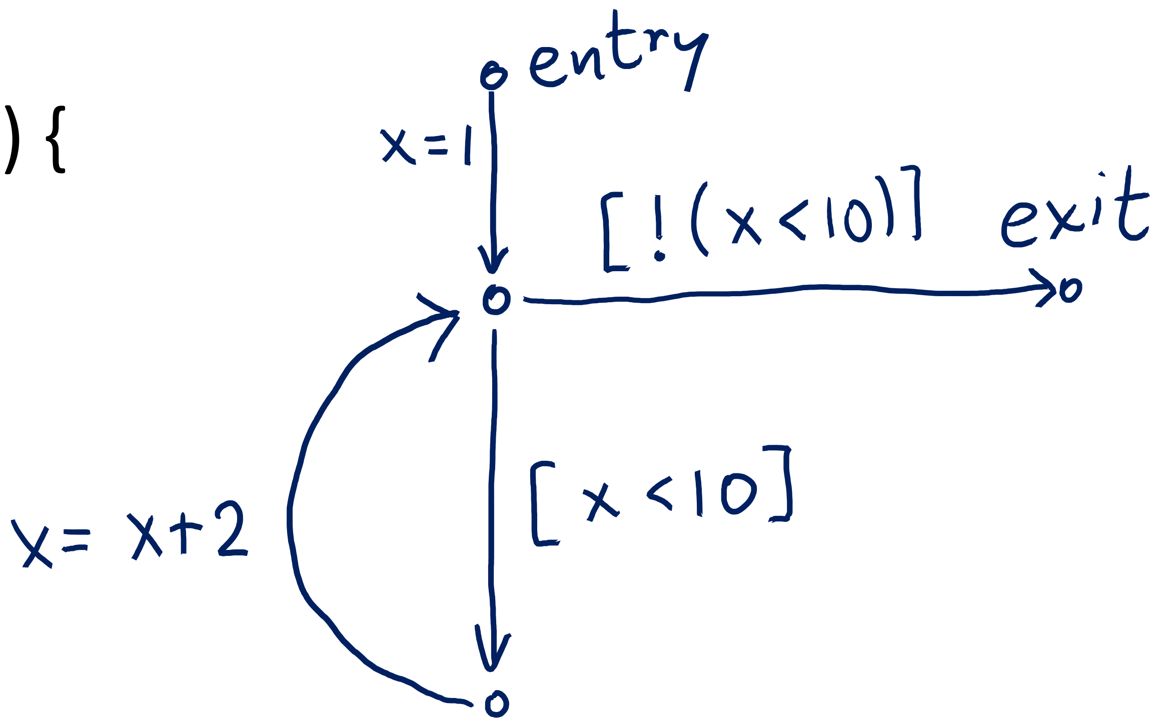
Handling Loops: Iterate Until Stabilizes

```
x = 1

while (x < 10) {

  x = x + 2

}
```



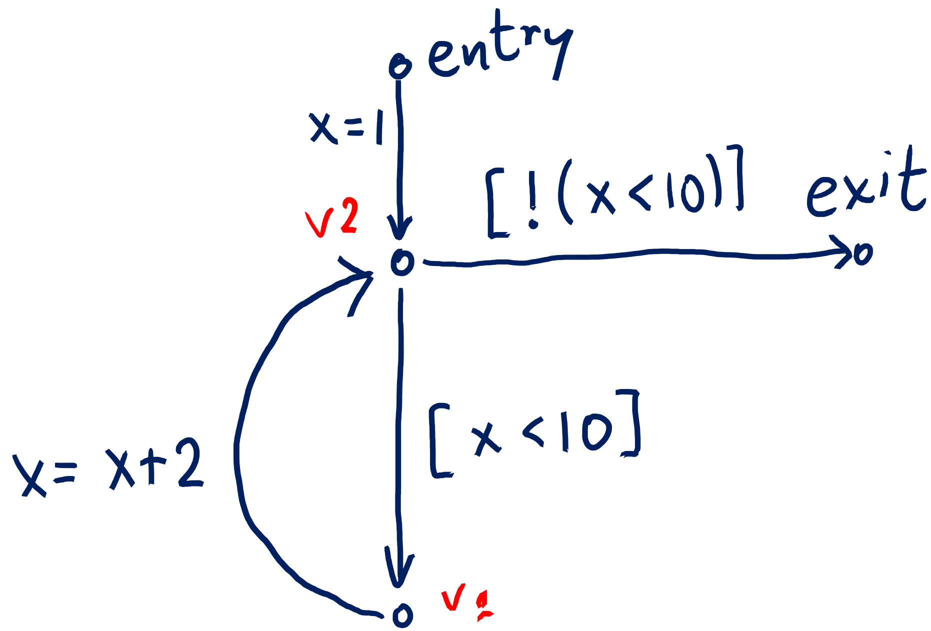
Analysis Algorithm

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** transferFun(statmt, facts(v1))
}

$\sqcup$

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

Exercise: Run range analysis, prove that **error** is unreachable

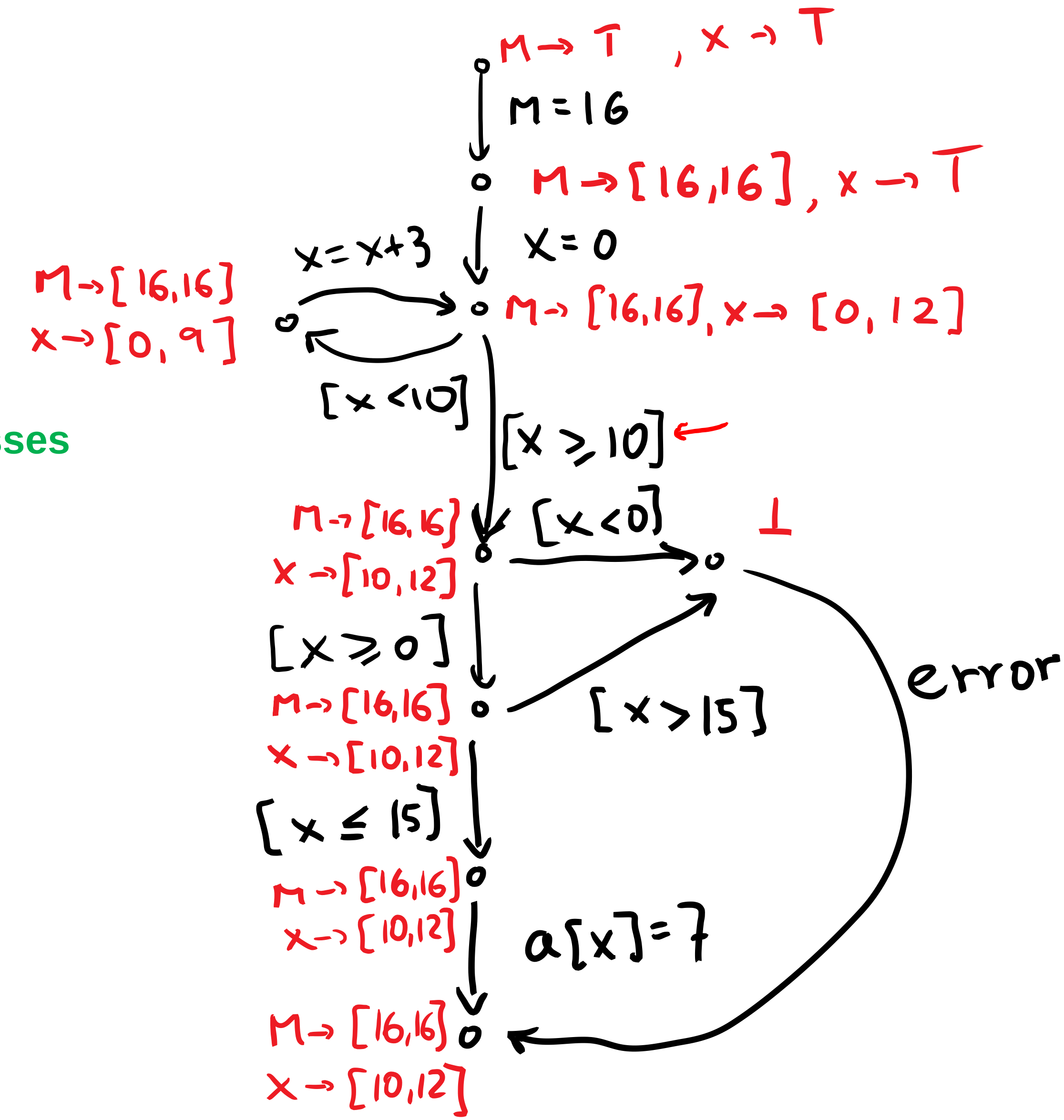
```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

checks array accesses

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;
}
```

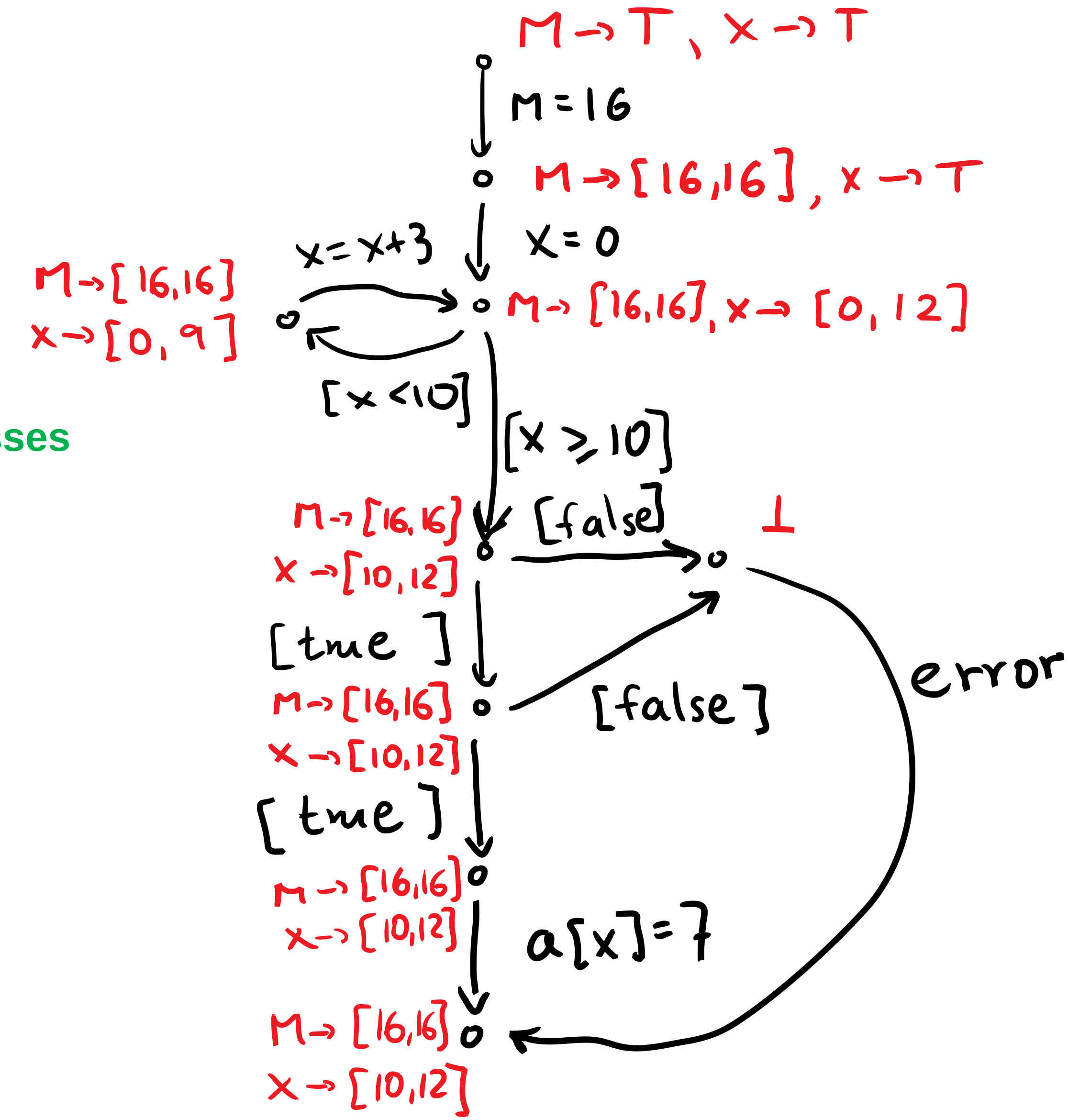
checks array accesses



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;
}
```

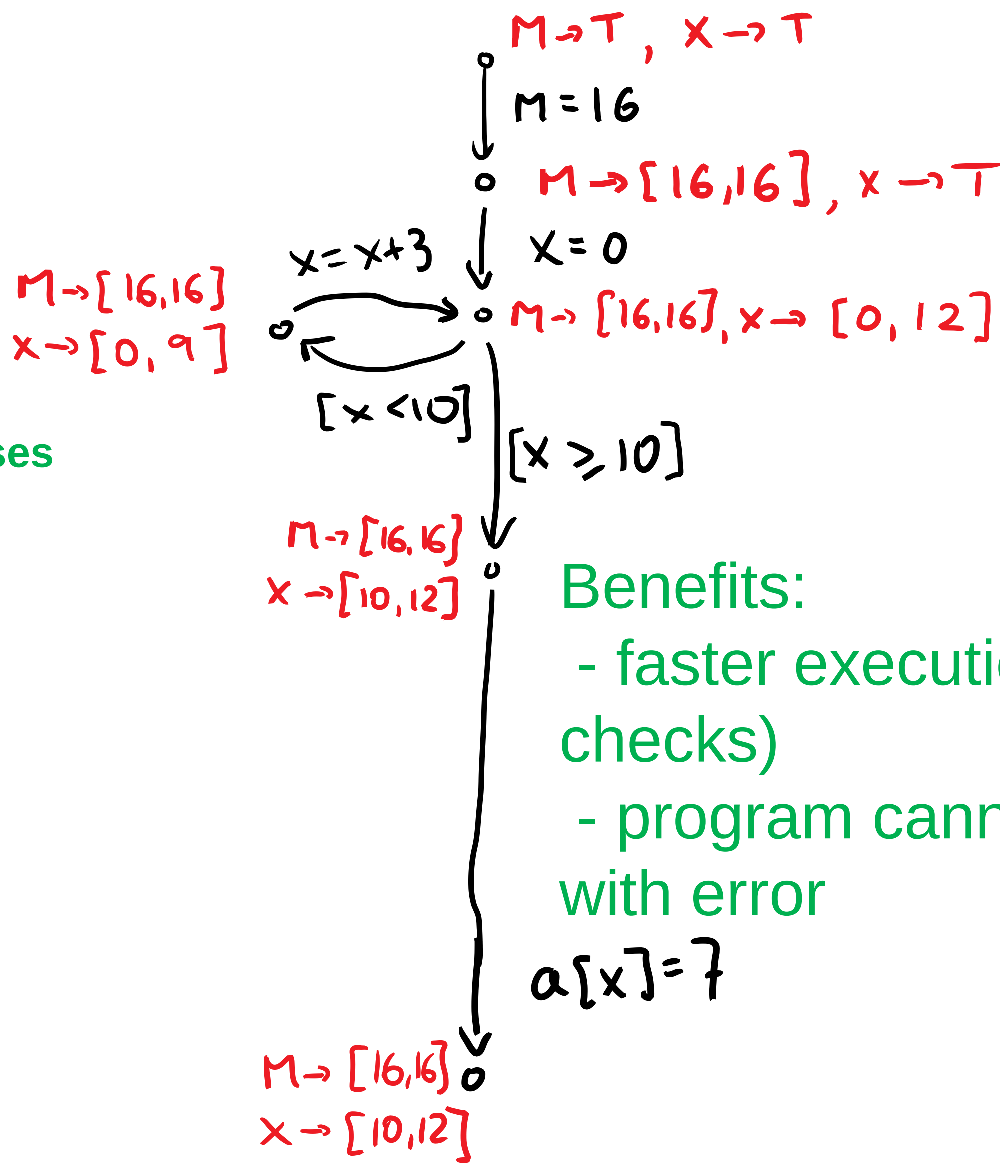
checks array accesses



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

Constant Propagation Domain

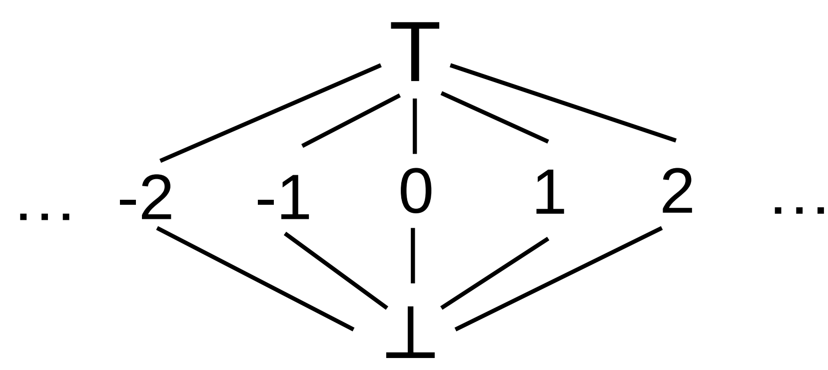
Domain values D are:

- intervals [a,a], denoted simply 'a'
- empty set, denoted ⊥ and set of all integers T

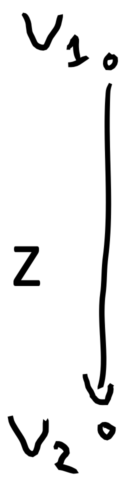
Formally, if **Z** denotes integers, then

$$D = \{\perp, T\} \cup \{ a \mid a \in \mathbf{Z}\}$$

D is an infinite set



Constant Propagation Transfer Functions



x = y + z

table for +:

z \ y	⊥	c_y	T
⊥	⊥	⊥	⊥
c_z	⊥	$c_y + c_z$	T
T	⊥	T	T

For **each variable** (x,y,z) and
each CFG node (program point)
we store: ⊥ , a constant, or T

```
abstract class Element
case class Top extends Element
case class Bot extends Element
case class Const(v:Int) extends Element
var facts : Map[Nodes,Map[VarNames,Element]]
  what executes during analysis of x=y+z:
oldY = facts(v1)("y")
oldZ = facts(v1)("z")
newX = tableForPlus(oldY, oldZ)
facts(v2) = facts(v2) join facts(v1).updated("x", newX)
```

```
def tableForPlus(y:Element, z:Element)
= (x,y) match {
  case (Const(cy),Const(cz)) =>
                         Const(cy+cz)
  case (Bot,_) => Bot
  case (_,Bot) => Bot
  case (Top,Const(cz)) => Top
  case (Const(cy),Top) => Top
}
```

Run Constant Propagation

What is the number of updates?

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

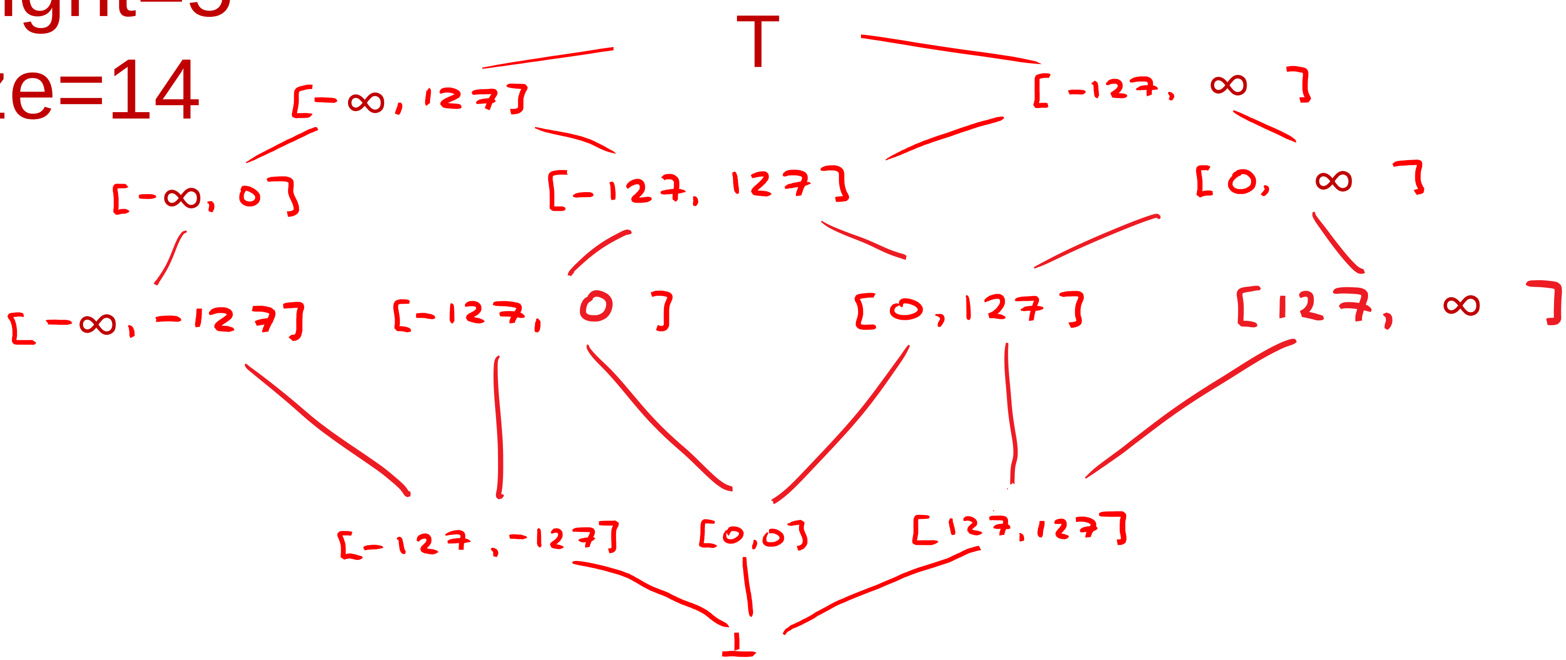
```
x = 1
n = readInt()
while (x < n) {
  x = x + 2
}
```

Observe

- Range analysis with end points
  W = {-128, 0, 127} has a finite domain
- Constant propagation has infinite domain
  (for every integer constant, one element)
- Yet, constant propagation finishes sooner!
  - it is not about the size of the domain
  - it is about the height

Height of Lattice: Length of Max. Chain

height=5
size=14



height=2
size =∞