Abstract Interpretation

(Cousot, Cousot 1977)

also known as Data-Flow Analysis

(Kildall 1973)

Why Constant Propagation

```
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) {
  print(i);
  i = i + step;    // can emit decrement
  if (step > 0) {
    c = (i < b);
  } else {
    c = (i > a); // can emit better instruction here
  } // insert here (a = a + step), redo analysis
}
```

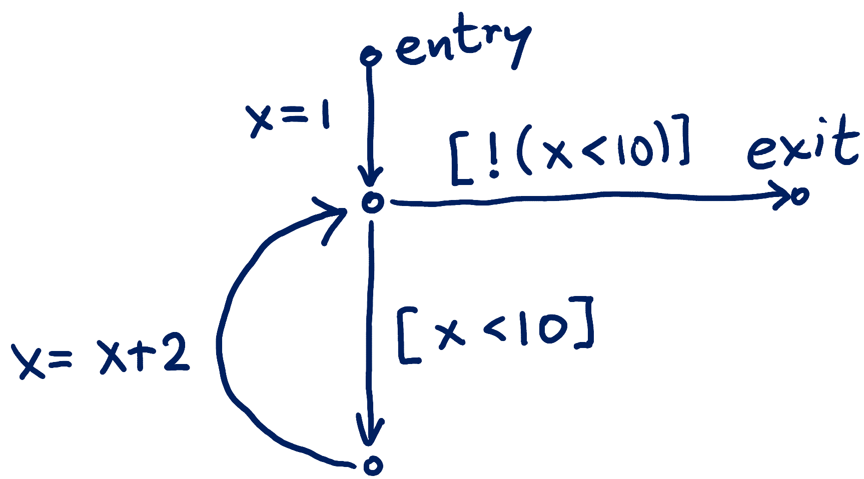
Goal of Data-Flow Analysis

Automatically compute information about the program

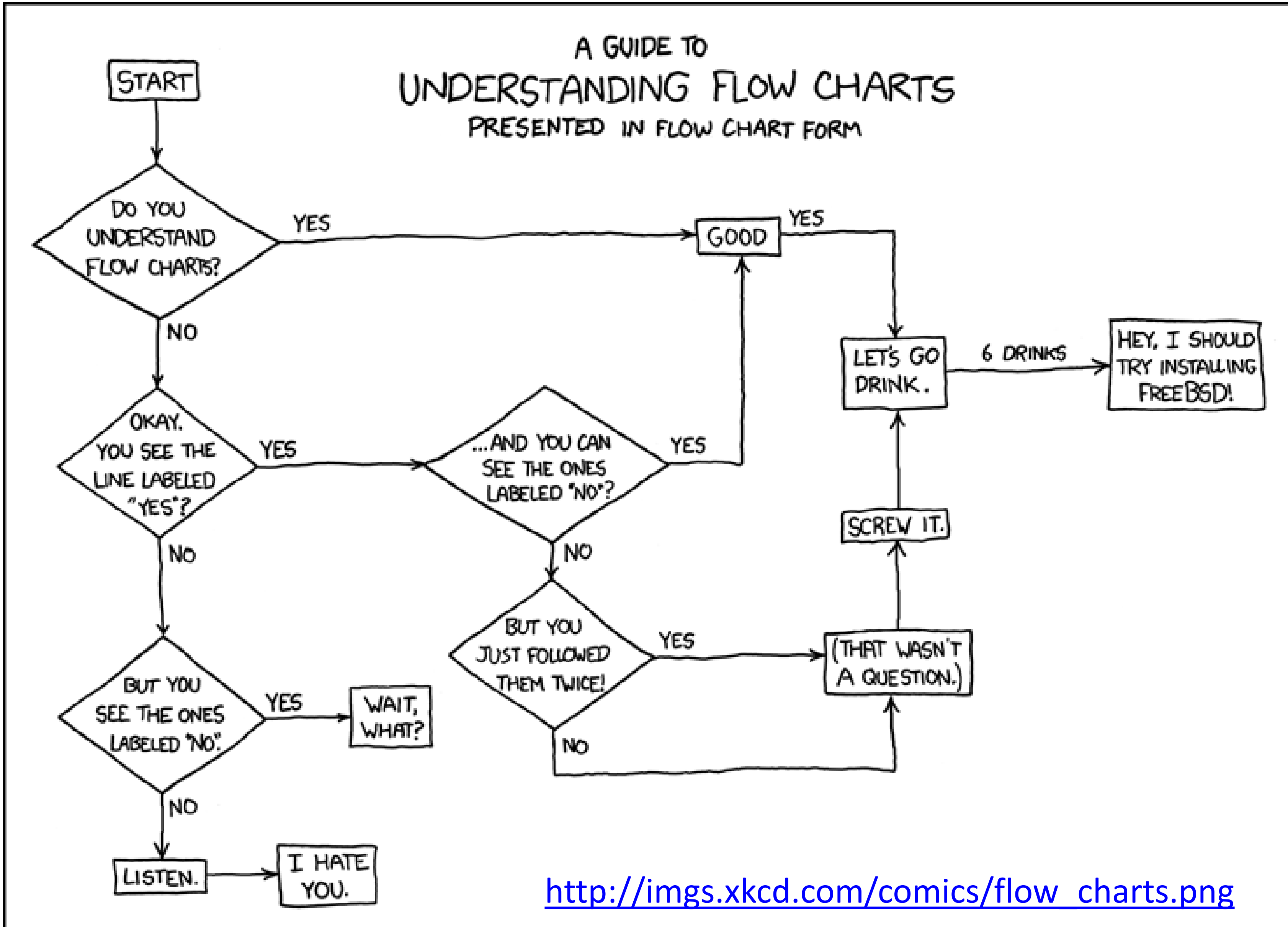
- Use it to report errors to user (like type errors)
- Use it to optimize the program

Works on control-flow graphs:
(like flow-charts)

```
x = 1
while (x < 10) {
  x = x + 2
}
```



Control-Flow Graphs: Like Flow Charts



http://imgs.xkcd.com/comics/flow_charts.png

Control-Flow **Graph**: (V,E)

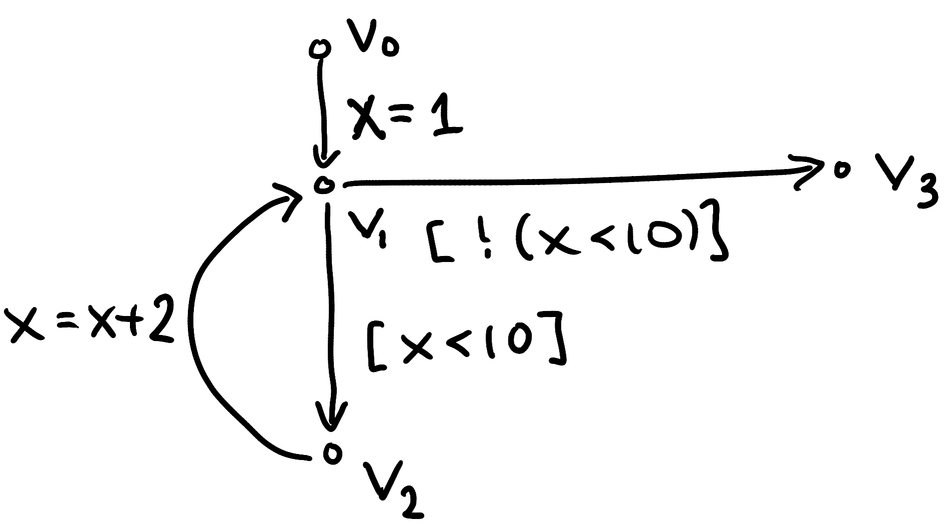
Set of nodes, V

Set of edges, which have statements on them

$$(v_1,st,v_2) \in E$$

means there is edge from v_1 to v_2 labeled with statement st.

x = 1
while (x < 10) {
x = x + 2
}



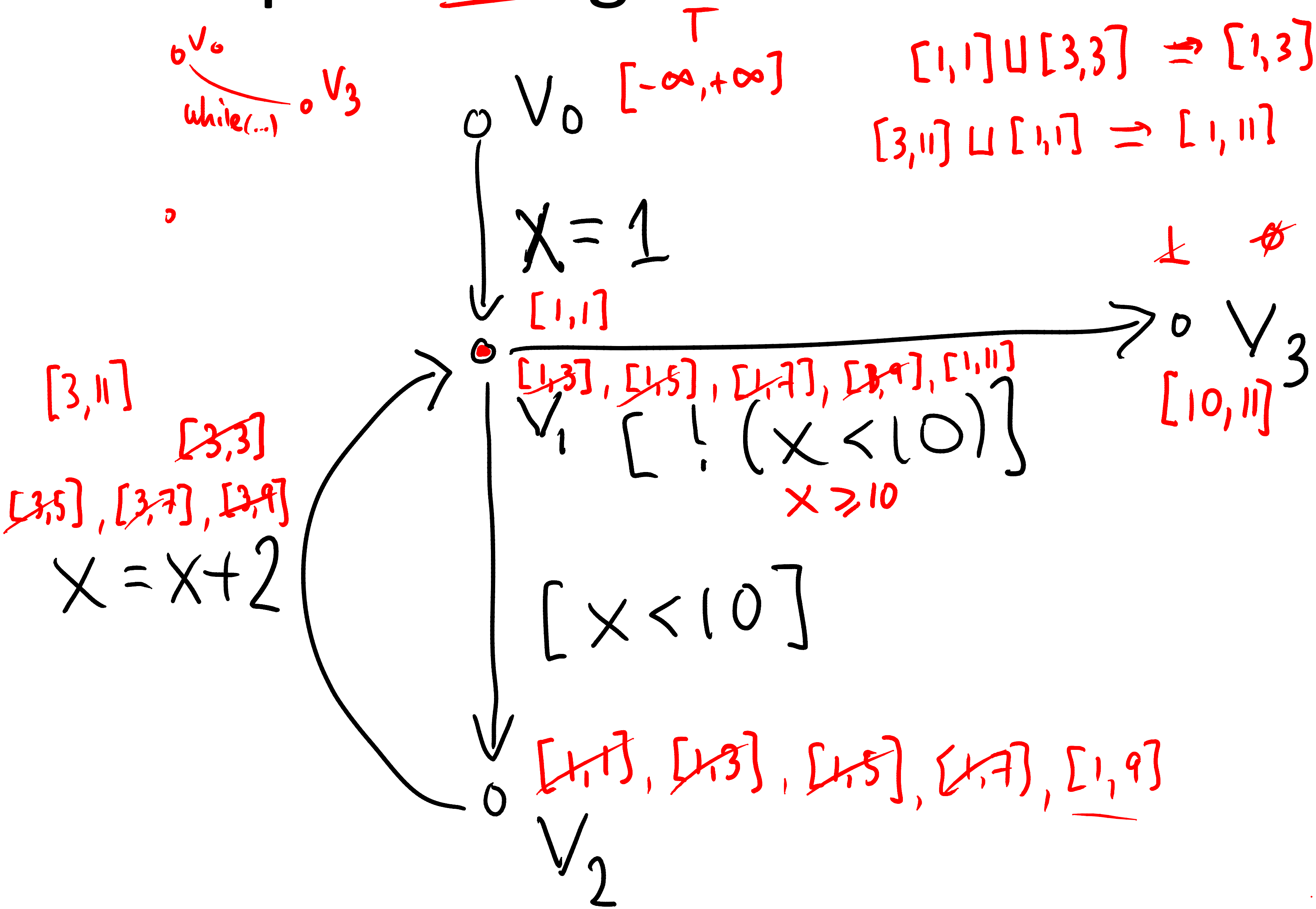
$V = \{v_0,v_1,v_2,v_3\}$

$E = \{(v_0,x=1,v_1), (v_1,[x<10],v_2),$
$(v_2,x=x+2,v_1), (v_1,[!(x<10)],v_3)\}$

Interpretation and Abstract Interpratation

- Control-Flow graph is similar to AST
- We can
  - interpret control flow graph
  - generate machine code from it (e.g. LLVM, gcc)
  - abstractly interpret it: do not push values, but **approximately compute supersets of possible values** (e.g. intervals, types, etc.)

Compute Range of x at Each Point



What we see in the sequel

1. How to compile abstract syntax trees into control-flow graphs
2. Lattices, as structures that describe abstractly sets of program states (facts)
3. Transfer functions that describe how to update facts
4. Basic idea of fixed-point iteration

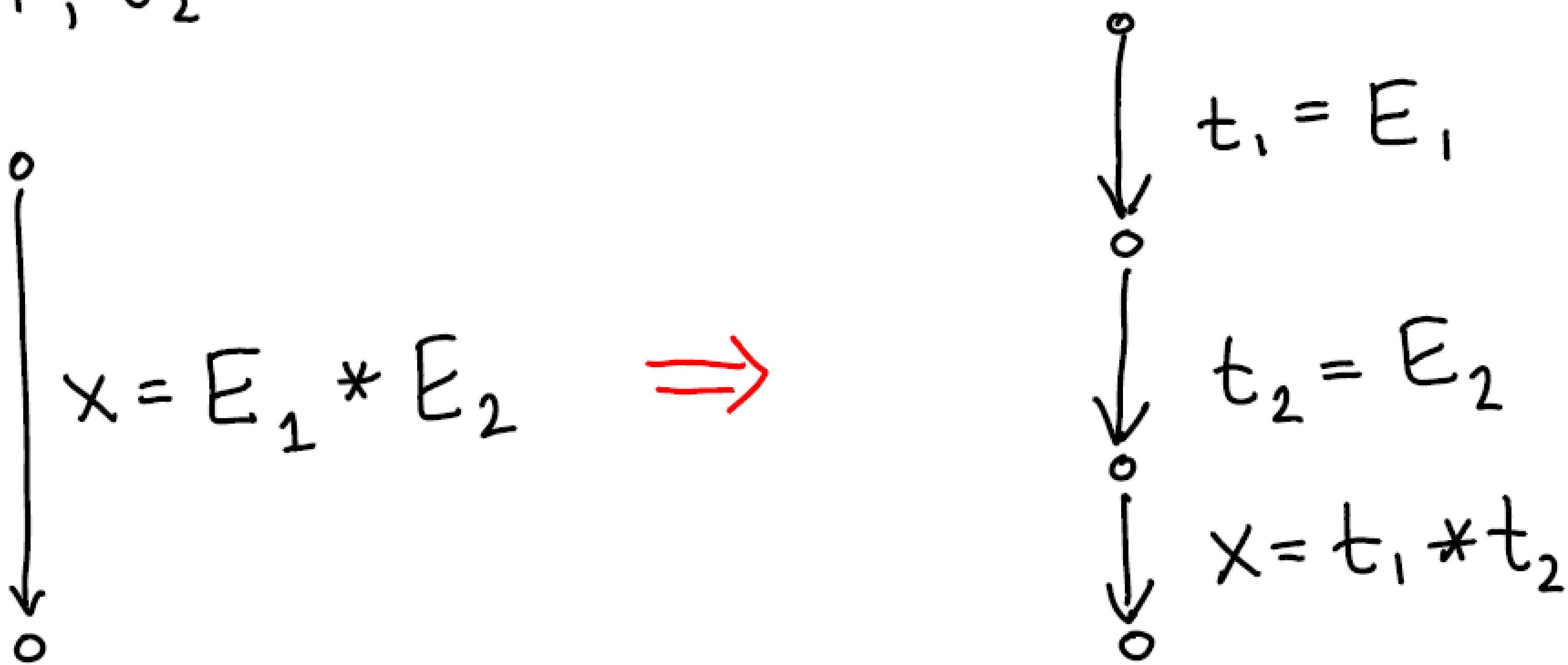
Generating Control-Flow Graphs

- Start with graph that has one entry and one exit node and label is entire program
- Recursively decompose the program to have more edges with simpler labels
- When labels cannot be decomposed further, we are done

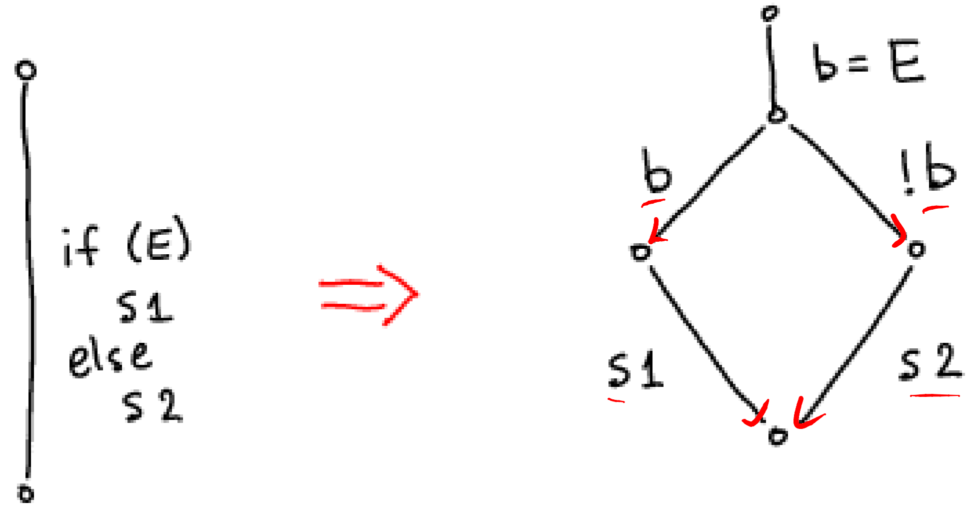
Flattening Expressions for simplicity and ordering of side effects

E_1, E_2 – complex expressions

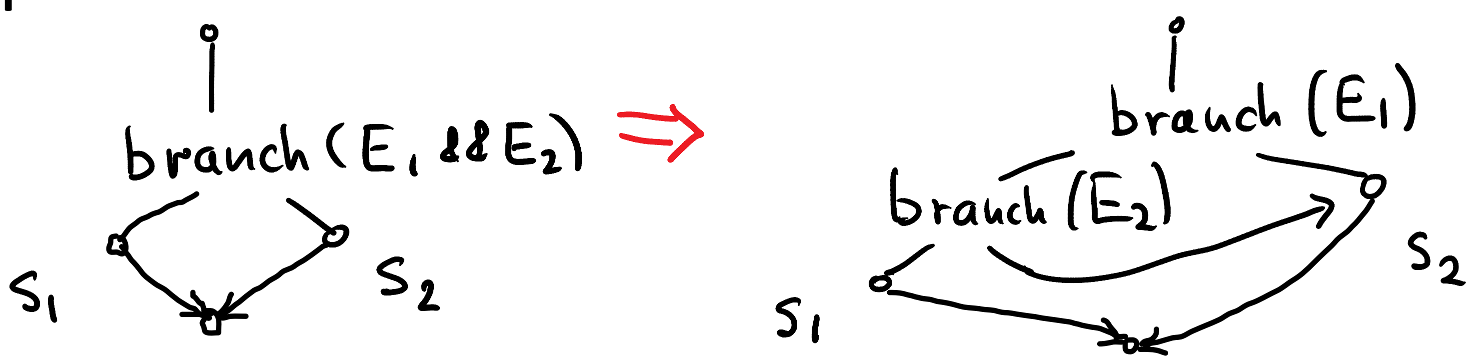
t_1, t_2 – fresh variables



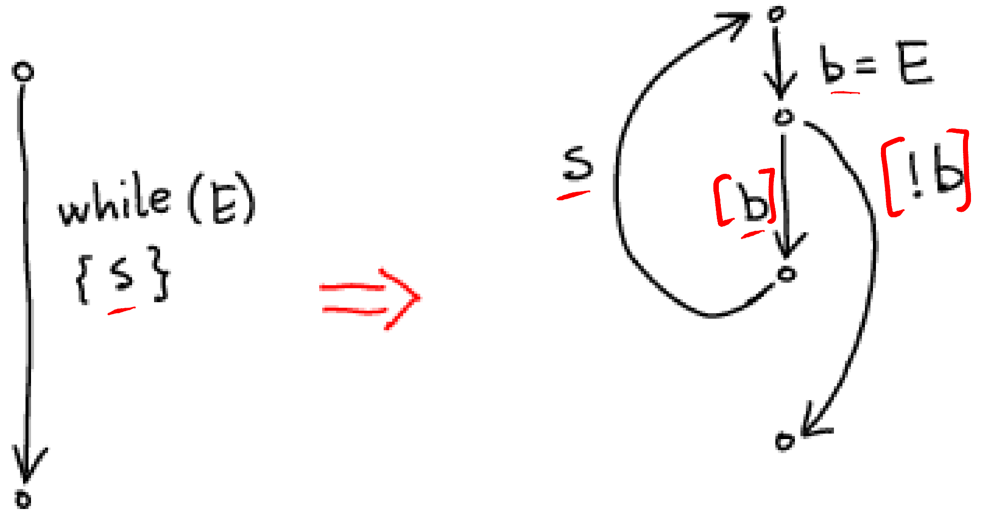
If-Then-Else



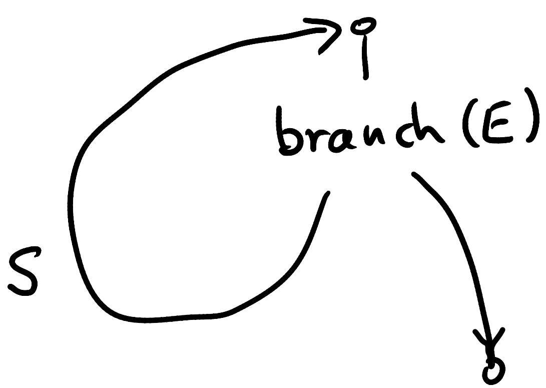
Better translation uses the "branch" instruction approach: have two destinations



While



Better translation uses the "branch" instruction



Example 1: Convert to CFG

v_0

while (i < 10) {

v_1

println(j);

v_2

i = i + 1;

v_3

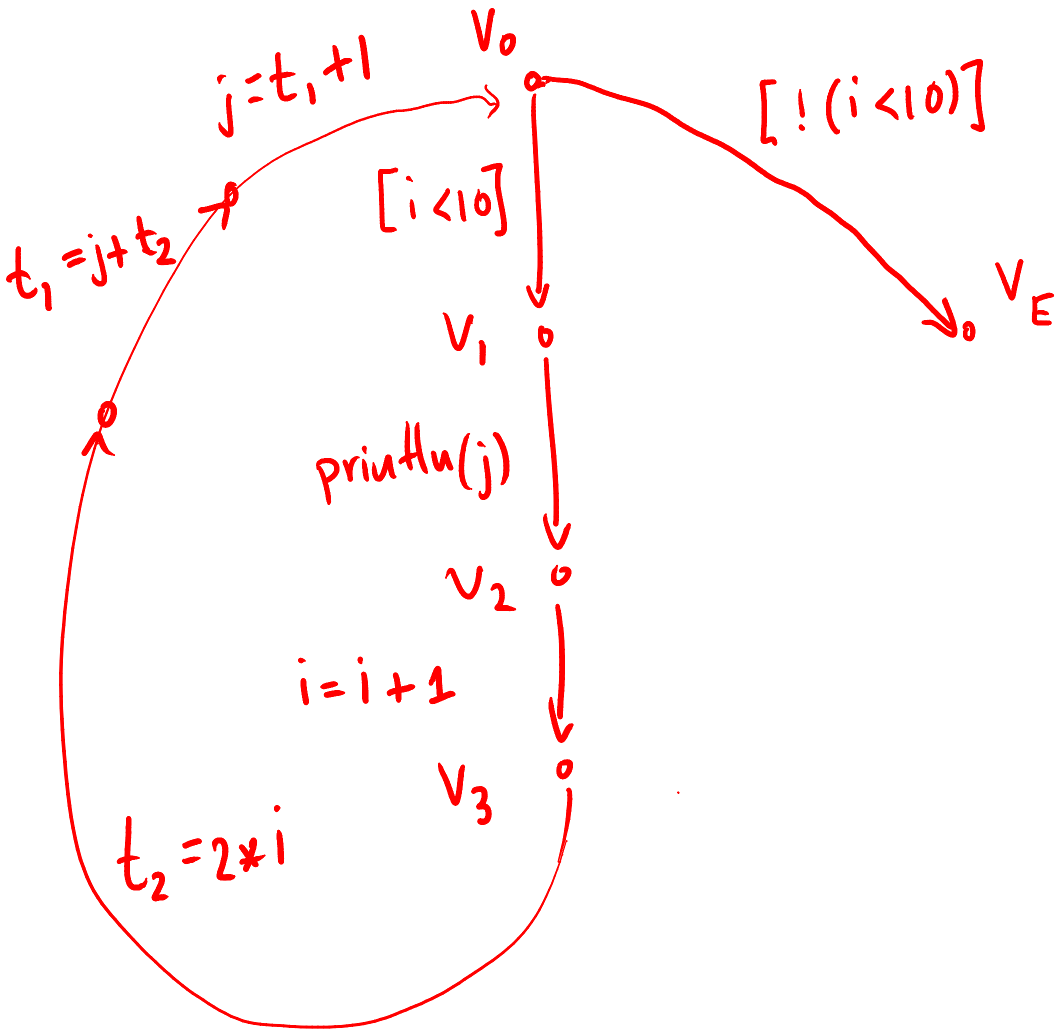
j = j +2*i + 1;

t_1

}

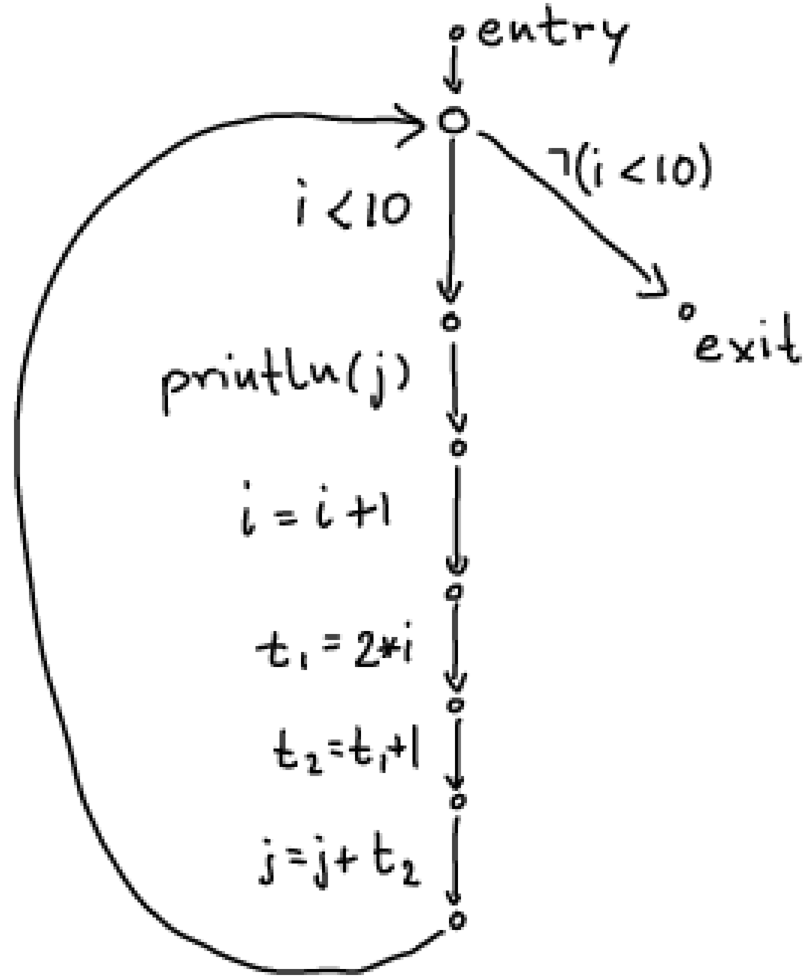
v_E

j=j+2*i+1



Example 1 Result

```
while (i < 10) {
  println(j);
  i = i + 1;
  j = j +2*i + 1;
}
```

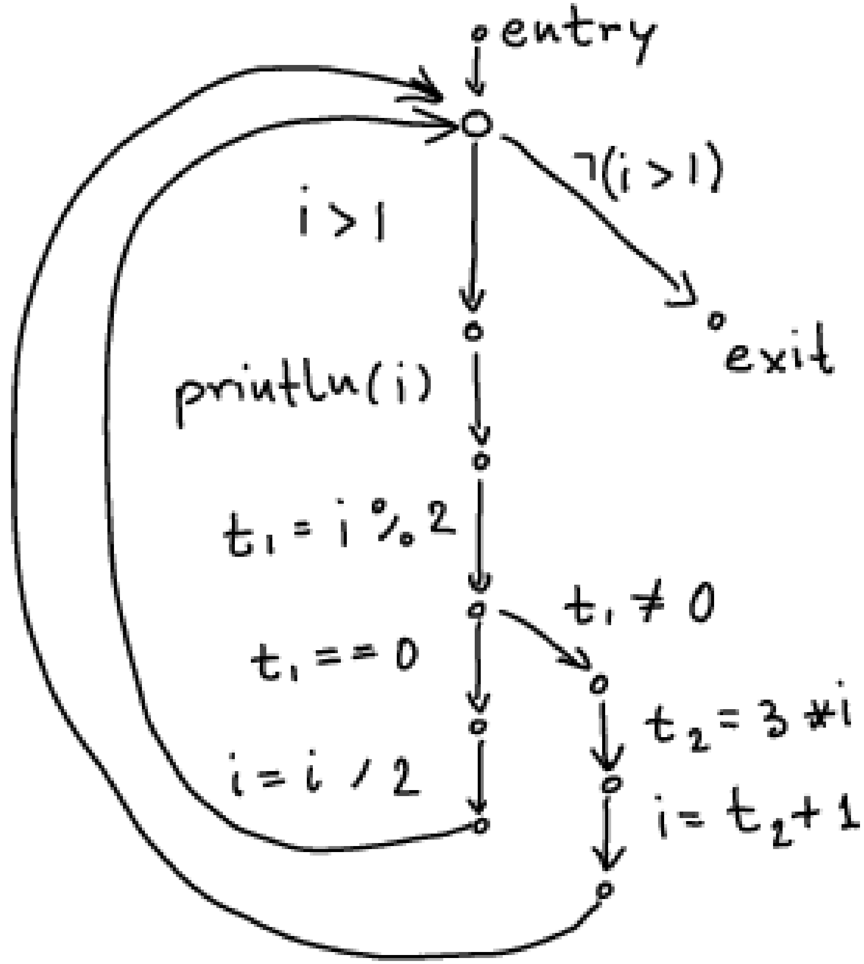


Example 2: Convert to CFG

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```

Example 2 Result

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```

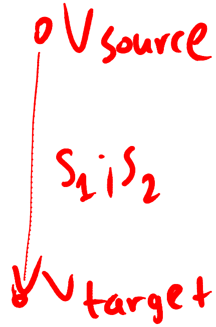


Translation Functions

$[\ s_1\ ;\ s_2\ ]\ v_{source}\ v_{target} =$

$\quad [\ s_1\ ]\ v_{source}\ v_{fresh}$

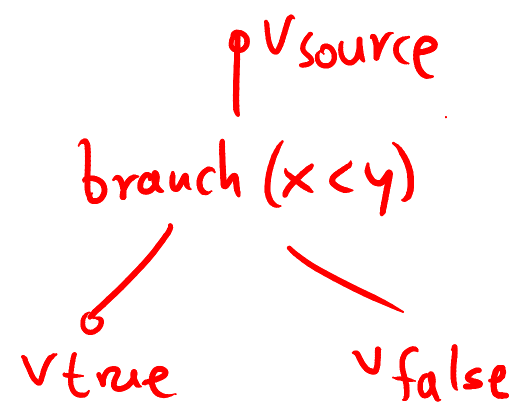
$\quad [\ s_2\ ]\ v_{fresh}\ v_{target}$



$[$ **branch**$(x<y)\ ]\ v_{source}\ v_{true}\ v_{false} =$

insert$(v_{source}, [x<y], v_{true})$;

insert$(v_{source}, [!(x<y)], v_{false})$



insert $(v_s, stmt, v_t) =$

$cfg = cfg + (v_s, stmt, v_t)$

$[\ x=y+z\ ]\ v_s\ v_t =$ **insert**$(v_s, x=y+z, v_t)$

when y,z are constants or variables

Analysis Domain (D)
Lattices

Abstract Intepretation Generalizes Type Inference

Type Inference

- computes types
- type rules
  - can be used to compute types of expression from subtypes
- types fixed for a variable

Abstract Interpretation

- computes **facts** from a domain
  - types
  - intervals
  - formulas
  - set of initialized variables
  - set of live variables
- transfer functions
  - compute facts for one program point from facts at previous program points
- facts change as the values of vars change (*flow-sensitivity*)

scalac computes types. Try in REPL:

class C

class D **extends** C

class E **extends** C

val p = false

val d = **new** D()

val e = **new** E()

val z = **if** (p) d **else** e

val u = **if** (p) (d,e) **else** (d,d)

val v = **if** (p) (d,e) **else** (e,d)

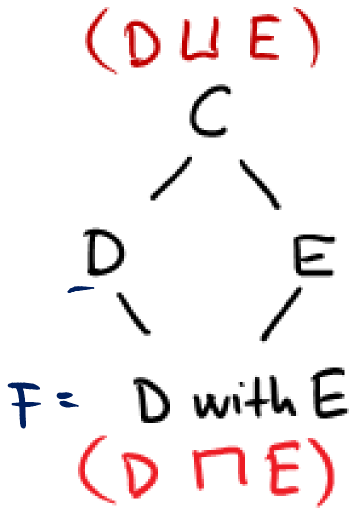
val f1 = **if** (p) ((d1:D) => 5) **else** ((e1:E) => 5)

val f2 = **if** (p) ((d1:D) => d) **else** ((e1:E) => e)

Finds "Best Type" for Expression

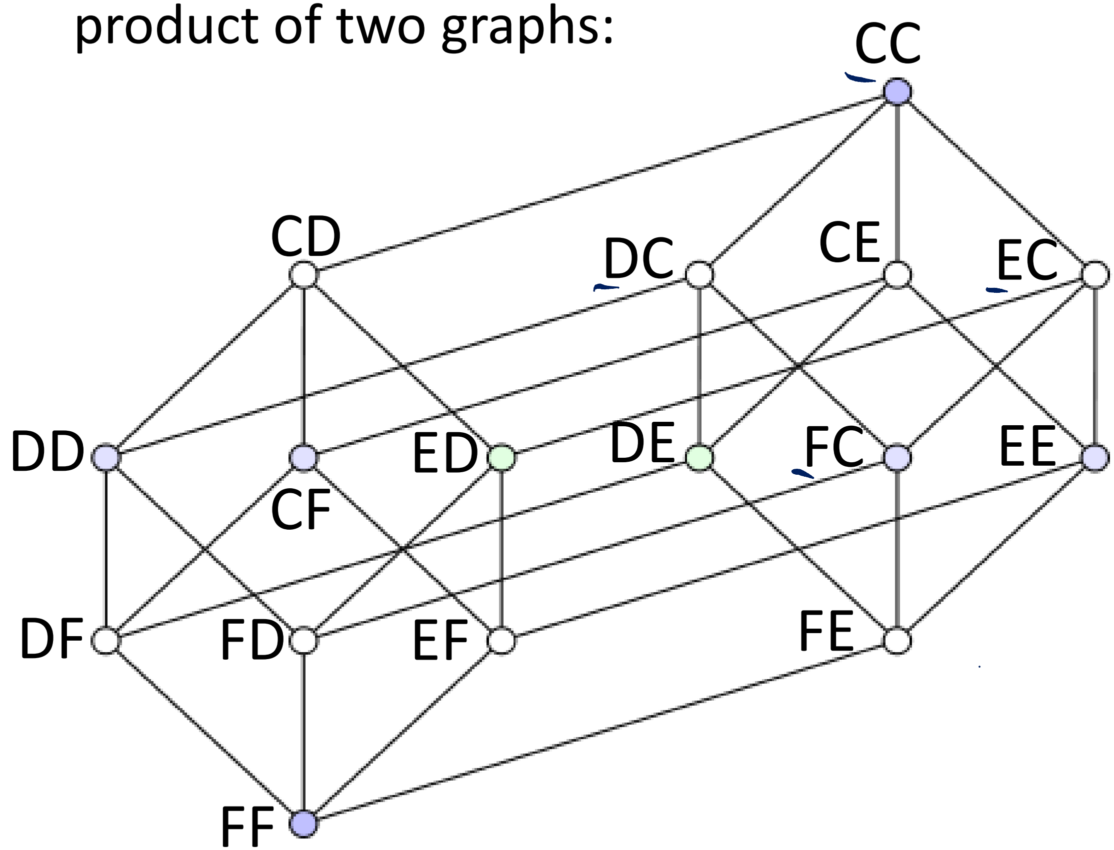
```
class C
class D extends C
class E extends C
val p = false
val d = new D()                                        //  d:D
val e = new E()                                        //  e:E
val z = if (p) d else e                                //  z:C

val u = if (p) (d,e) else (d,d)                        //  u:(D,C)
val v = if (p) (d,e) else (e,d)                        //  v:(C,C)

val f1 = if (p) ((d1:D) => 5) else ((e1:E) => 5)       //  f1: ((D with E) => Int)
val f2 = if (p) ((d1:D) => d) else ((e1:E) => e)       //  f2: ((D with E) => C)
```

Subtyping Relation in this Example



class C
class D **extends** C
class E **extends** C

product of two graphs:



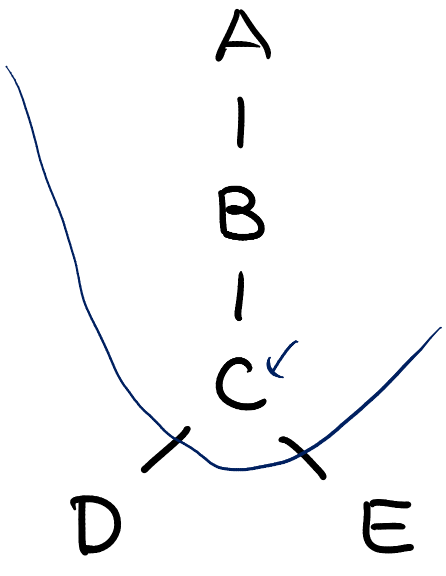
each relation can be visualized in 2D

– two relations: naturally shown in 4D (hypercube)

we usually draw larger elements higher

$\neg(x<y)$ $y \le x$ ← TOTAL ORDERS $\neg(x \le y)$

Least Upper Bound (lub, join)



A,B,C are all upper bounds on both D and E (they are above each of then in the picture, they are supertypes of D and supertypes of E).
Among these upper bounds, C is the least one (the most specific one).
We therefore say C is the **least upper bound**,

$$C = D \sqcup E$$

$[a,b] \sqcup [a',b'] = [\min(a,a'), \max(b,b')]$ $\subseteq$

In any partial order $\le$, if S is a set of elements (e.g. S={D,E}) then:

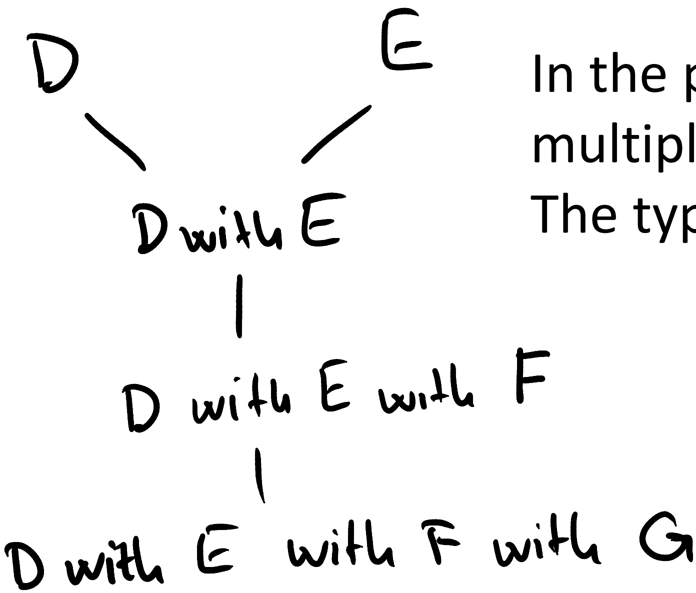
U is **upper bound** on S iff $x \le U$ for every x in S.

U_0 is the **least upper bound (lub)** of S, written $U_0 = \bigsqcup S$, or U_0=lub(S) iff:

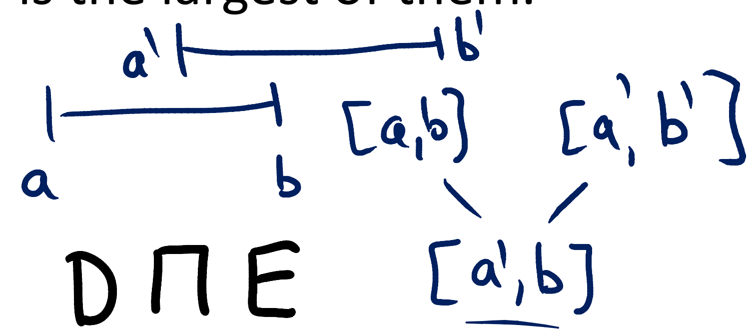
U_0 is upper bound and

if U is any upper bound on S, then $U_0 \le U$

Greatest Lower Bound (glb, meet)



In the presence of traits or interfaces, there are multiple types that are subtypes of both D and E. The type (D with E) is the largest of them.



In any partial order $\leq$, if S is a set of elements (e.g. S={D,E}) then:

L is **lower bound** on S iff L $\leq$ x for every x in S.

L_0 is the **greatest lower bound (glb)** of S, written $L_0 = \sqcup S$, or L_0=glb(S), iff:

m_0 is upper bound and

if m is any upper bound on S, then $m_0 \leq m$

Computing lub and glb for tuple and function types

$$(x_1, y_1) \sqcup (x_2, y_2) = (x_1 \sqcup x_2,\ y_1 \sqcup y_2)$$

$$(x_1, y_1) \sqcap (x_2, y_2) = (x_1 \sqcap x_2,\ y_1 \sqcap y_2)$$

$$(x_1 \to y_1) \sqcup (x_2 \to y_2) = (x_1 \sqcap y_1) \to (y_1 \sqcup y_2)$$

$$(x_1 \to y_1) \sqcap (x_2 \to y_2) = (x_1 \sqcup y_1) \to (y_1 \sqcap y_2)$$

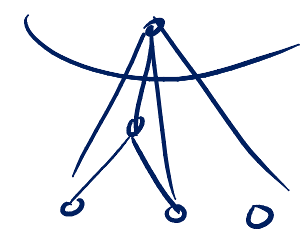
Lattice

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 **lub** and **glb**

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



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

Idea of Why Lemma Holds

- $\mathrm{lub}(x_1,\mathrm{lub}(x_2,\ldots,\mathrm{lub}(x_{n-1},x_n)))$ is $\mathrm{lub}(\{x_1,\ldots x_n\})$
- $\mathrm{glb}(x_1,\mathrm{glb}(x_2,\ldots,\mathrm{glb}(x_{n-1},x_n)))$ is $\mathrm{glb}(\{x_1,\ldots x_n\})$
- lub of all elements in D is maximum of D
  - by definition, glb({}) is the maximum of D
- glb of all elements in D is minimum of D
  - by definition, lub({}) is the minimum of D

$$\bigsqcup \emptyset = \bot = \sqcap \overbrace{\{x_1,\ldots,x_m\}}^{D}$$

$$\sqcap \emptyset = \top = \bigsqcup \underbrace{\{x_1,\ldots,x_m\}}_{D}$$

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

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:

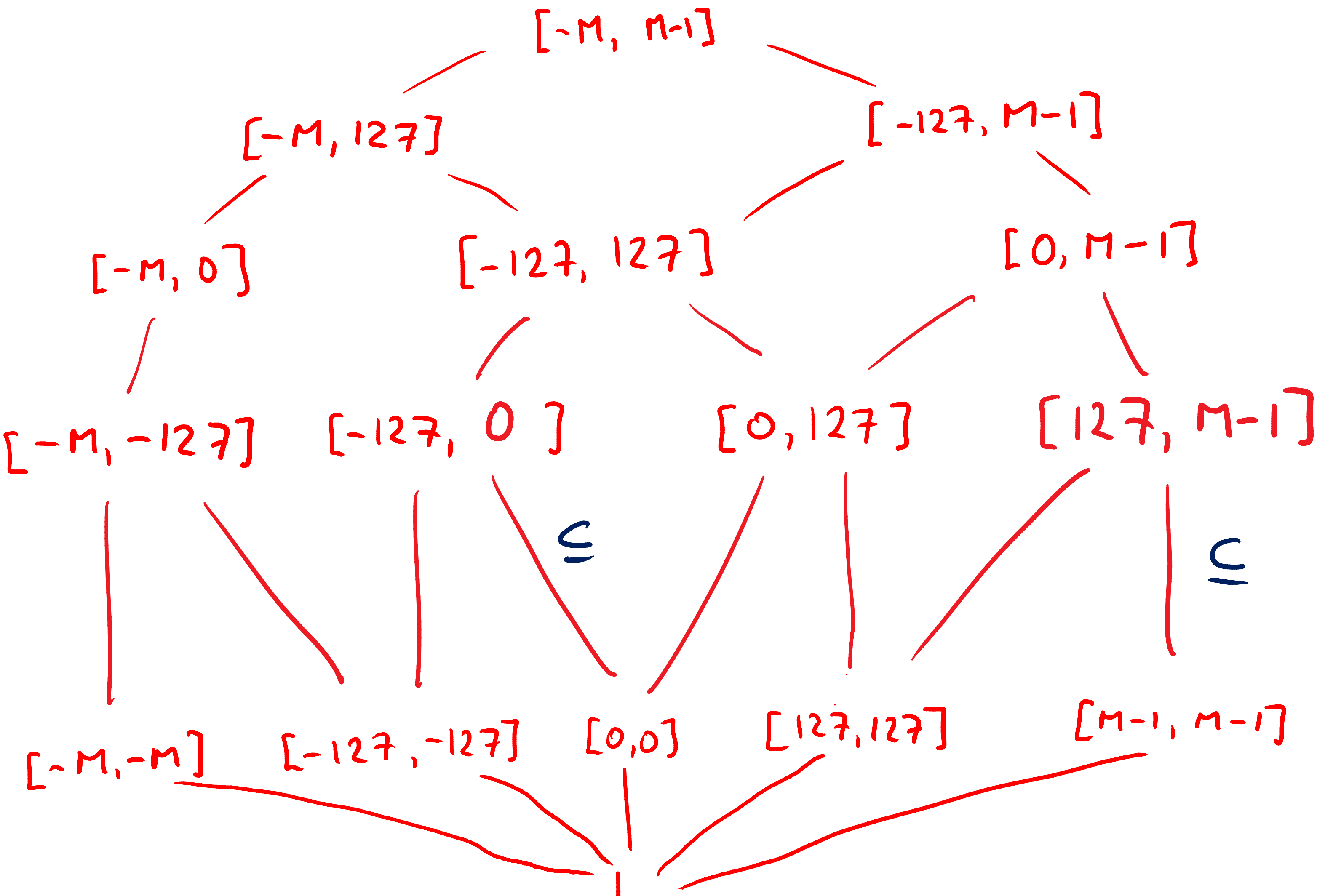
$[\![x = x+2]\!](x:[a,b], \ldots)$
$= (x:[a+2, b+2], \ldots)$

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

x = x+2

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

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



For now, we consider arbitrary integer bounds for intervals

- Really 'Int' should be BigInt, as in Haskell, Go
- 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
- For now, 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

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

$\downarrow$ $x = x+y$

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

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

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

Further transfer functions

- $x=y*z$ (assigning product)

$y: [a,b]$ $z: [c,d]$

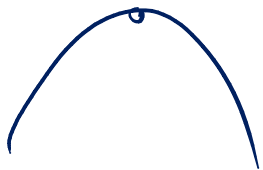
$\downarrow$

$y: [a',b']$ $z: [c',d']$

$S = \{ y \cdot z \mid y \in [a,b],\ z \in [c,d] \}$

min(S)

max(S)



- $x=y$ (copy)

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

$\downarrow$ $x=y$

$x: [c,d]$ $y: [c,d]$

Transfer Functions for **Tests**

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

$x : [-10, 10]$

if (x > 1) { $x \geq 2$

$x : [2, 10]$

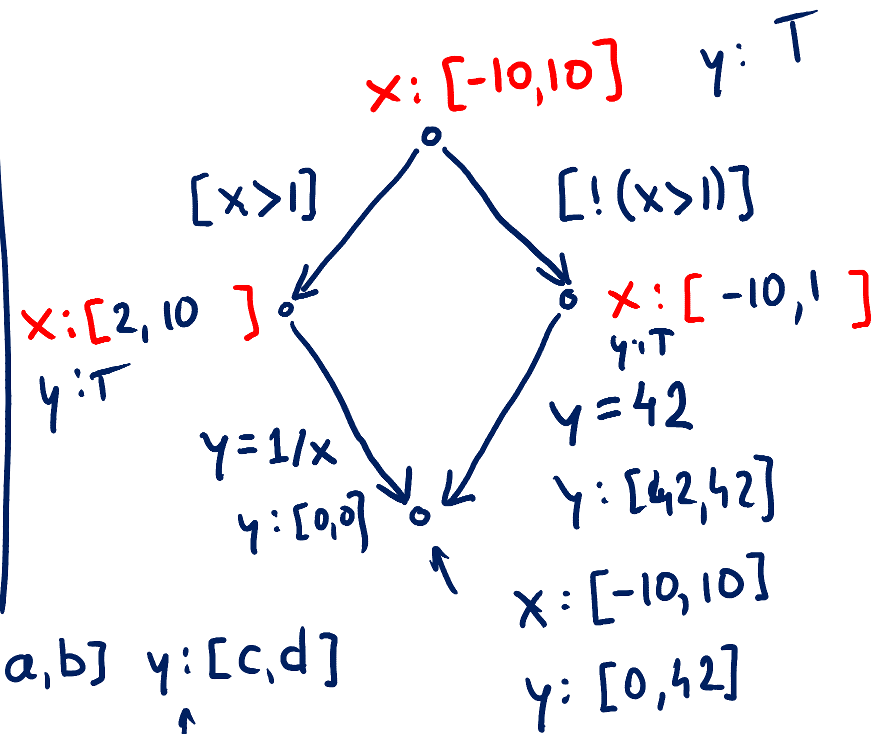
y = 1 / x ← $y : [0, 0]$

} else { $x \leq 1$

$x : [-10, 1]$

y = 42 $y : [42, 42]$

}



$x \geq y+1$

$y \geq c$

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

$[x > y]$

$y \leq x - 1 \leq b - 1$

$x : [\boxed{\max(a, c+1)}, b] \quad y : [c, \min(d, b-1)]$

Joining Data-Flow Facts

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

```
if (x > 0) {
```

x: y:

```
  y = x + 100
```

x: y:

```
} else {
```

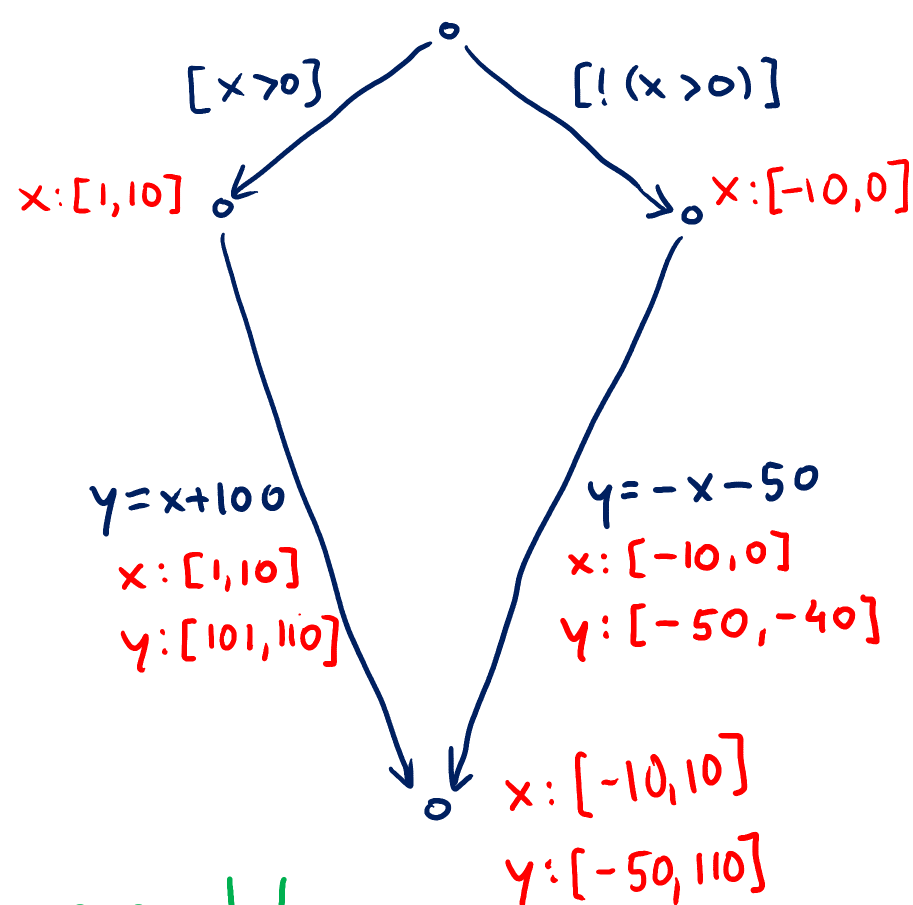
x: y:

```
  y = -x – 50
```

x: y:

```
}
```

x: y:



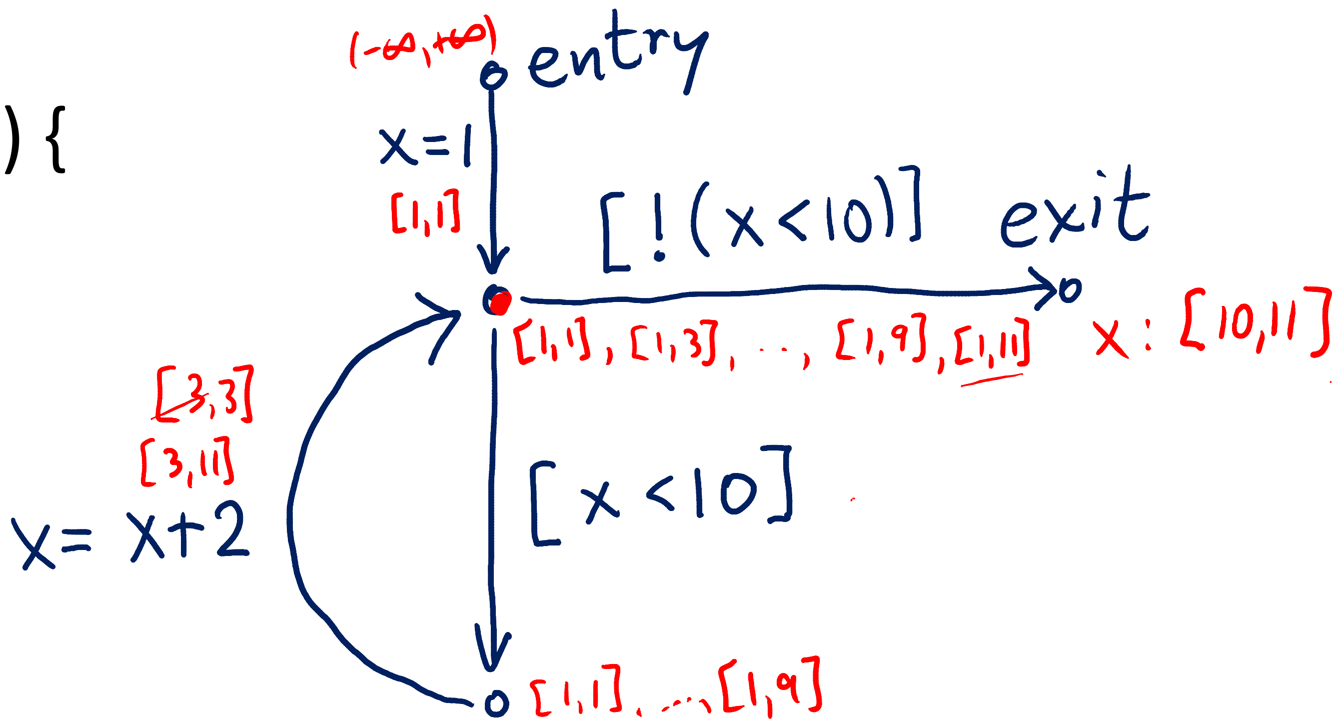
join $\sqcup$

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

Handling Loops: Iterate Until Stabilizes

```
x = 1

while (x < 10) {

  x = x + 2

}
```

x = 11

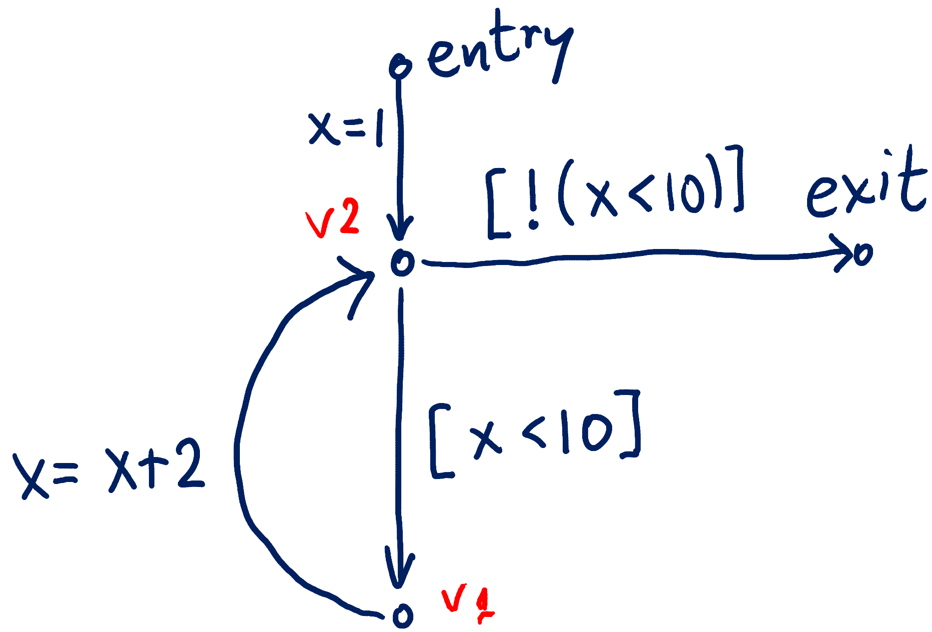


Analysis Algorithm

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

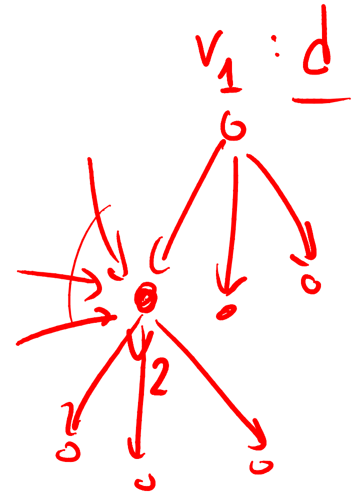
while (there was change)
  pick edge (v1,statmt,v2) from CFG
      such that facts(v1) has changed
  facts(v2)=facts(v2) **join** transferFun(statmt, facts(v1))
}

$\sqcup$



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

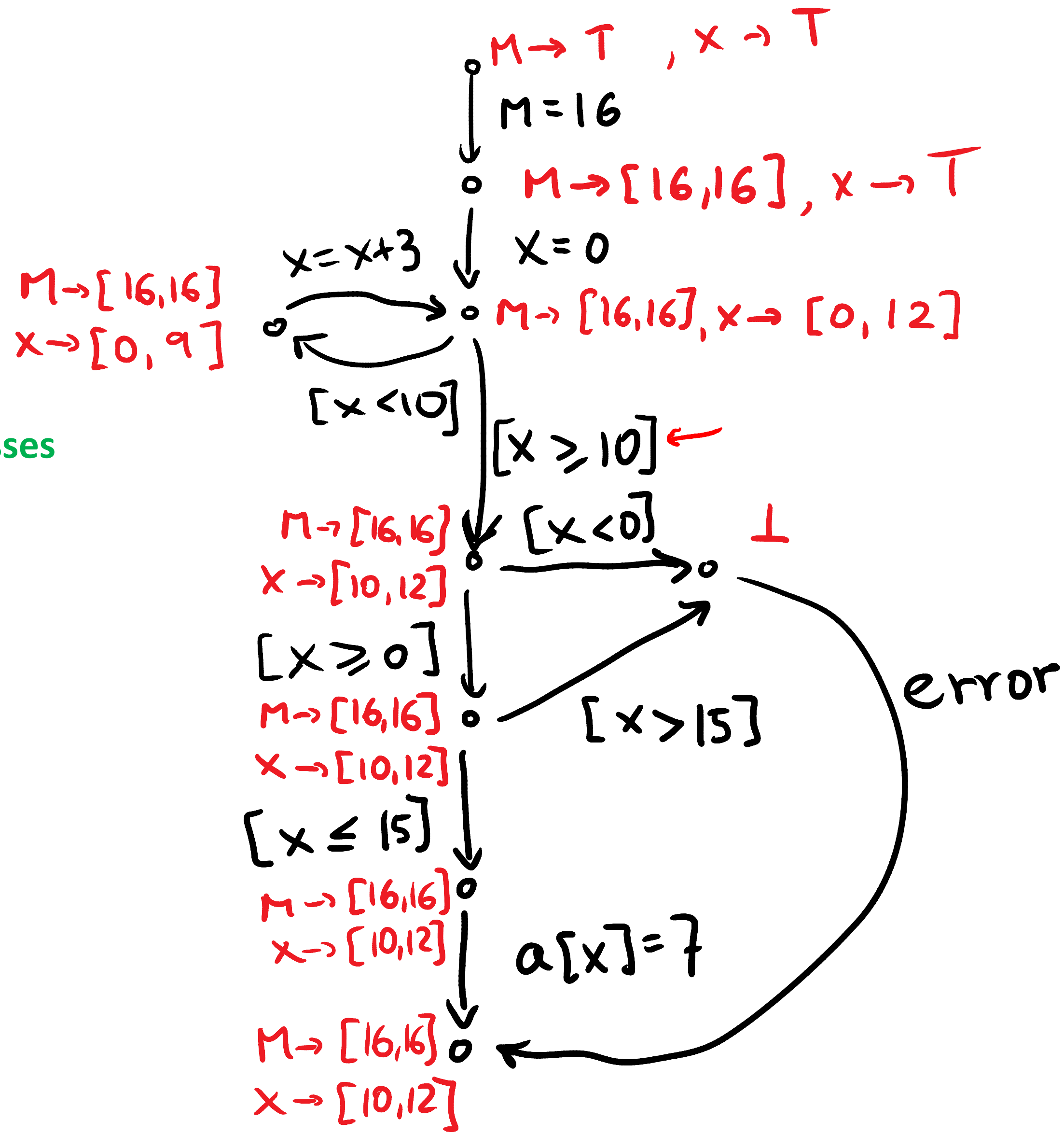
```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

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;
}
```

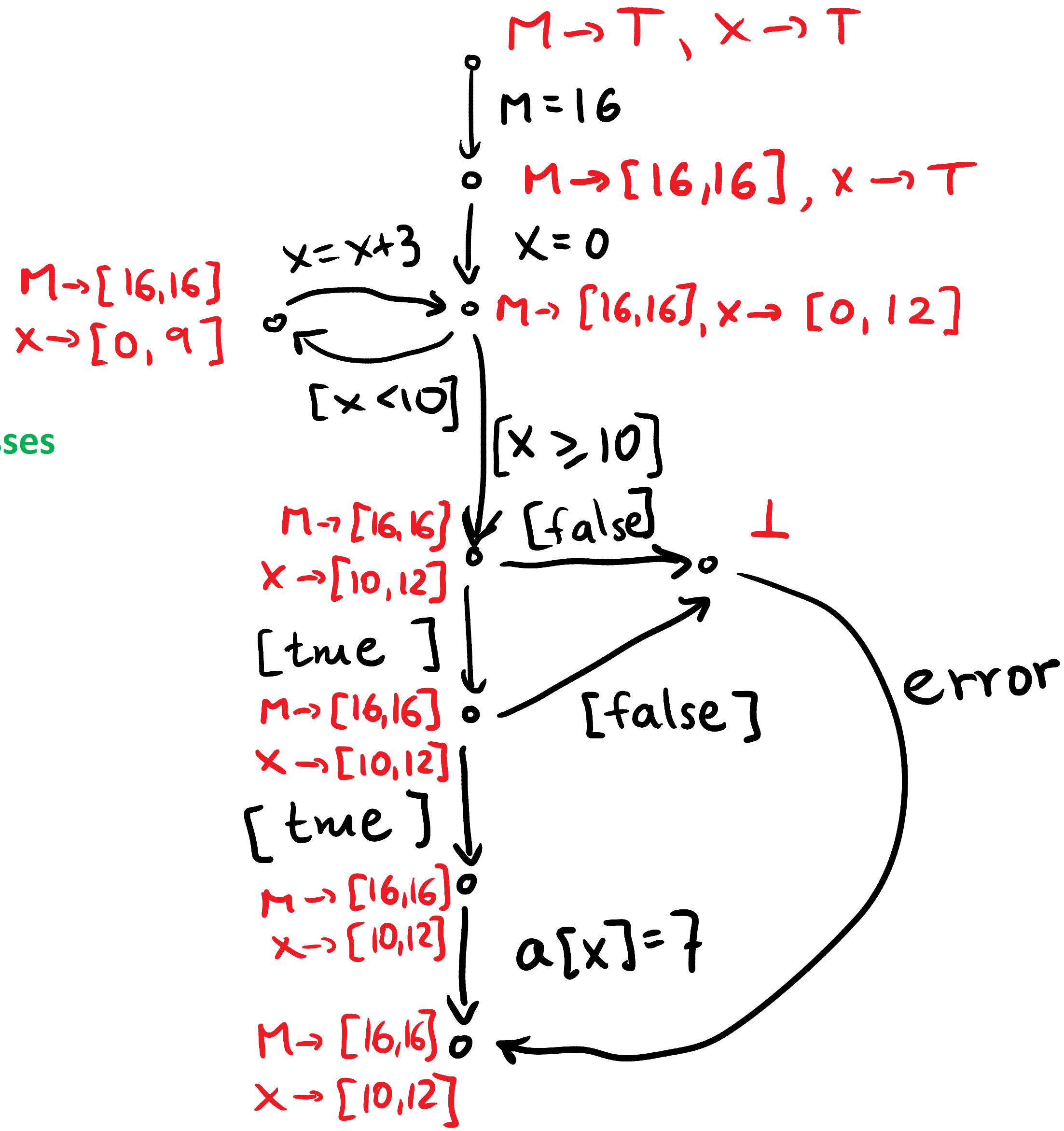
checks array accesses



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;
}
```

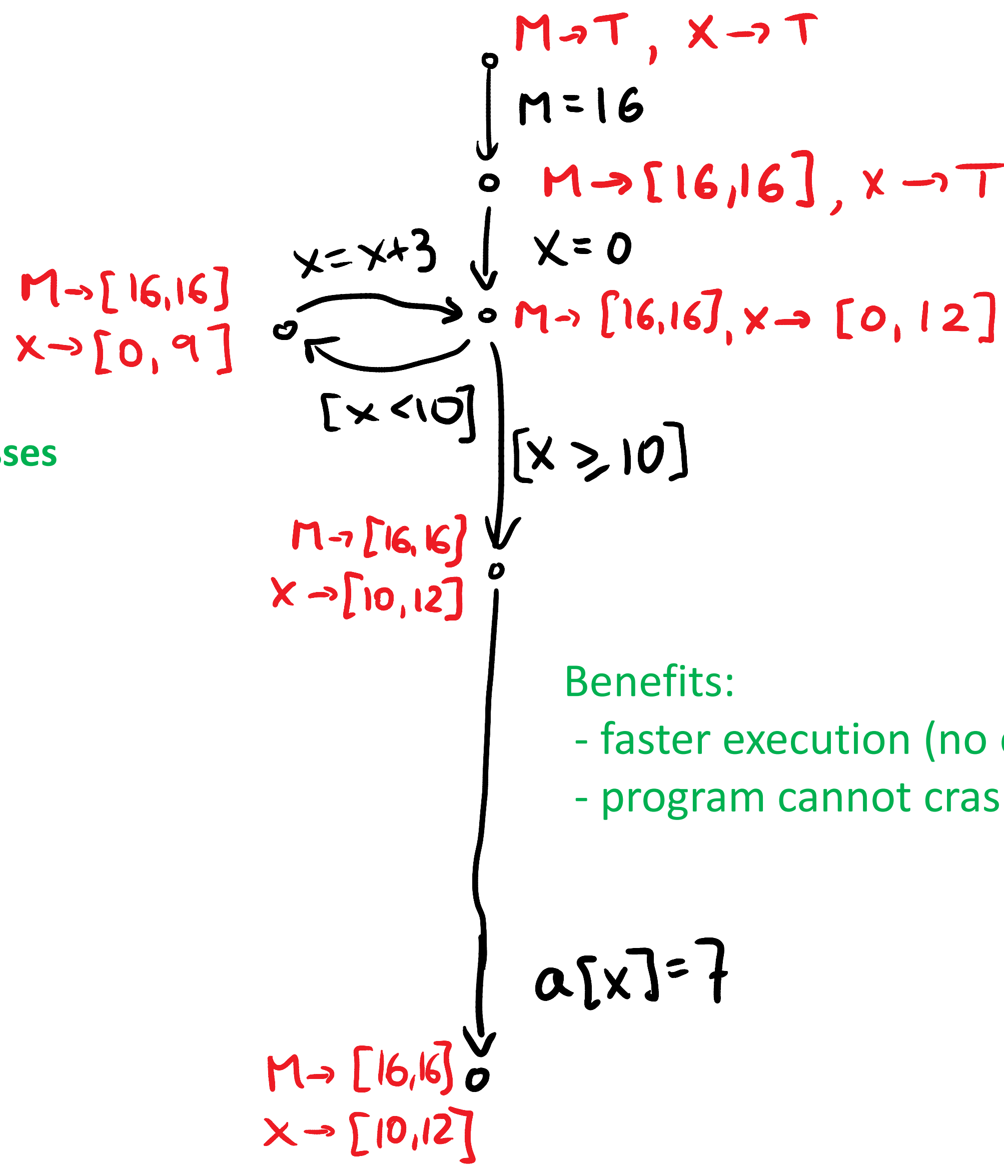
checks array accesses



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

Exercise: Apply Range Analysis and Simplify

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

$\perp$ $\top$

```
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);
  }
}
```