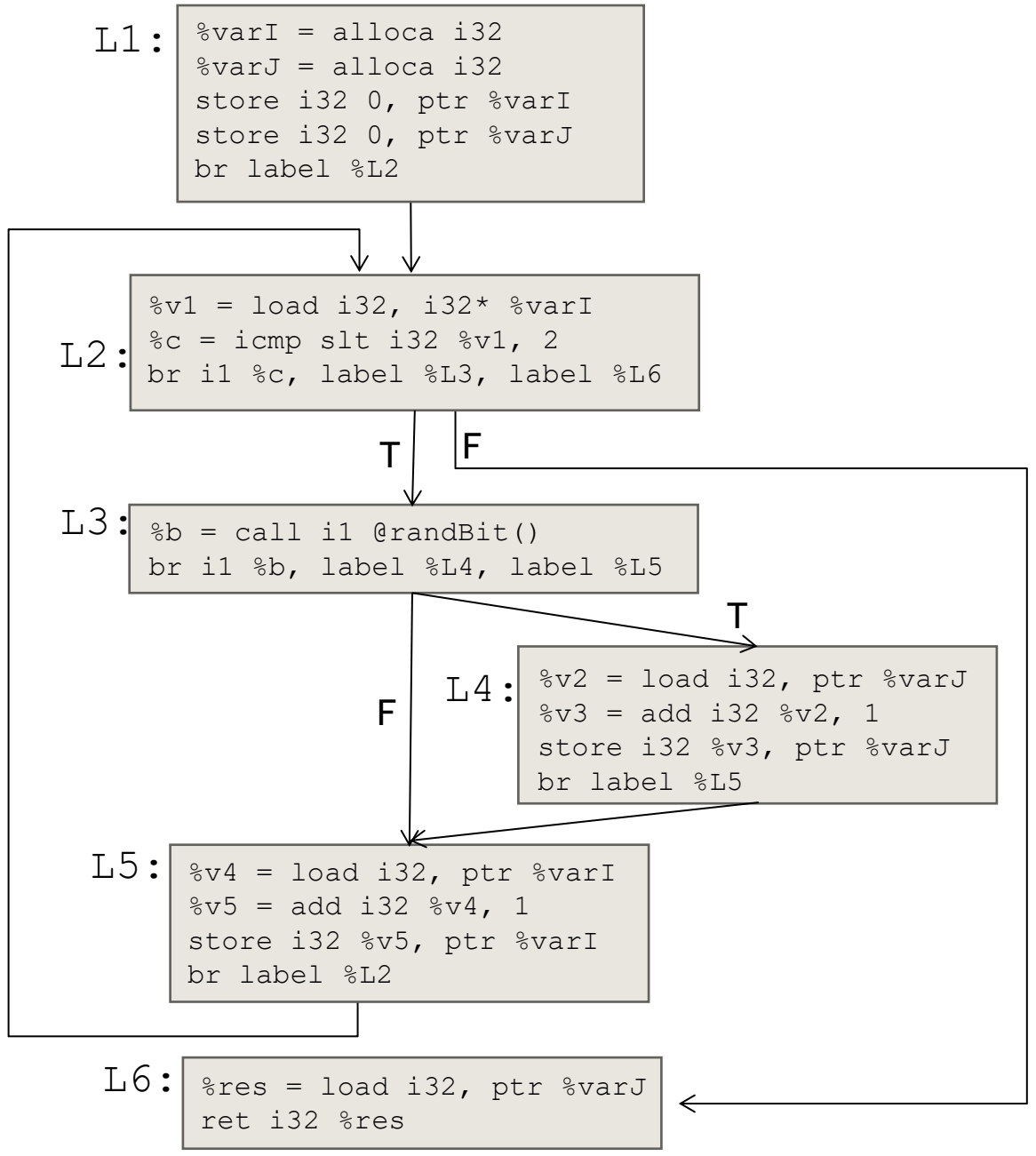
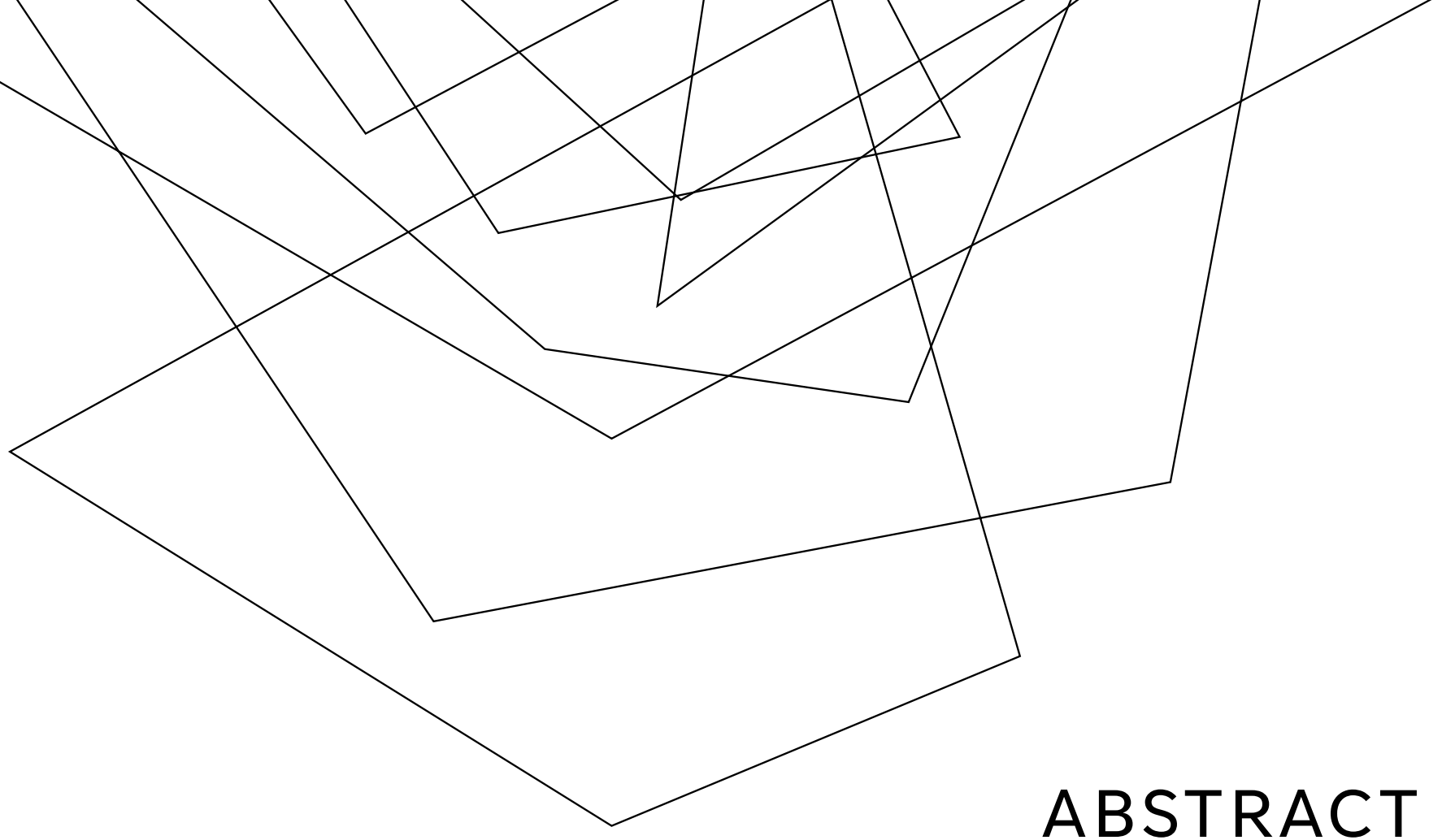
EXERCISE 10

Consider the CFG to the right, roughly equivalent to the C code below.
Indicate the value-set computed for `*varJ` at the end of L6 using the flow-sensitive saturation algorithm described in the last lecture

```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```



ADMINISTRIVIA AND ANNOUNCEMENTS



ABSTRACT INTERPRETATION

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: SATURATION

REVIEW: STATIC ANALYSIS

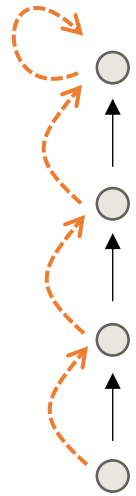
EXTENDING OUR BASIC DATAFLOW TO LOOPS

- No obvious start-point for analysis (circular dependence)

  Initializer values
- No obvious end-point (can't necessarily do with a single pass)

  Run the algorithm until the sets saturate

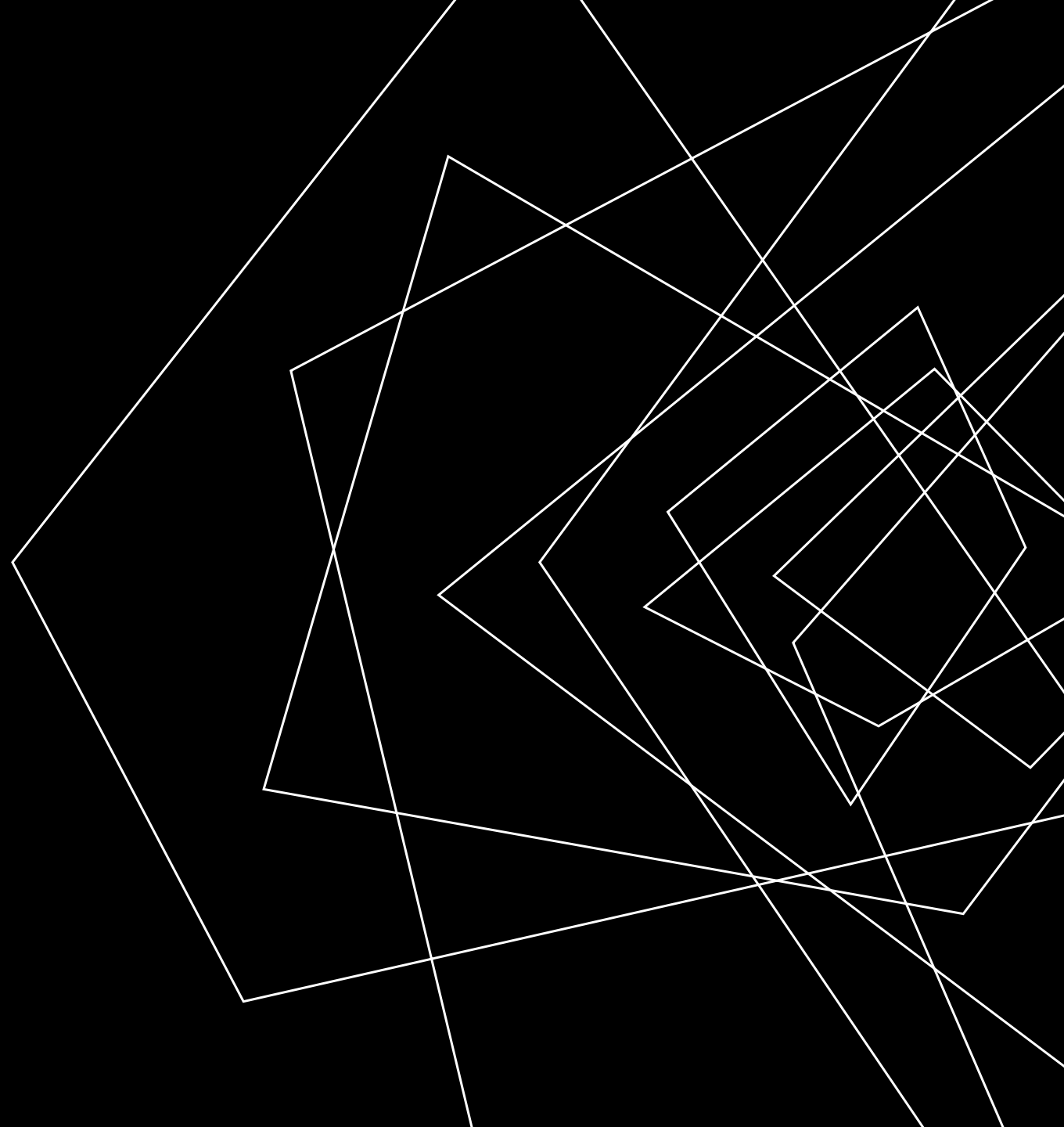
TERMINATE AT SATURATION POINT (AKA FIXPOINT)



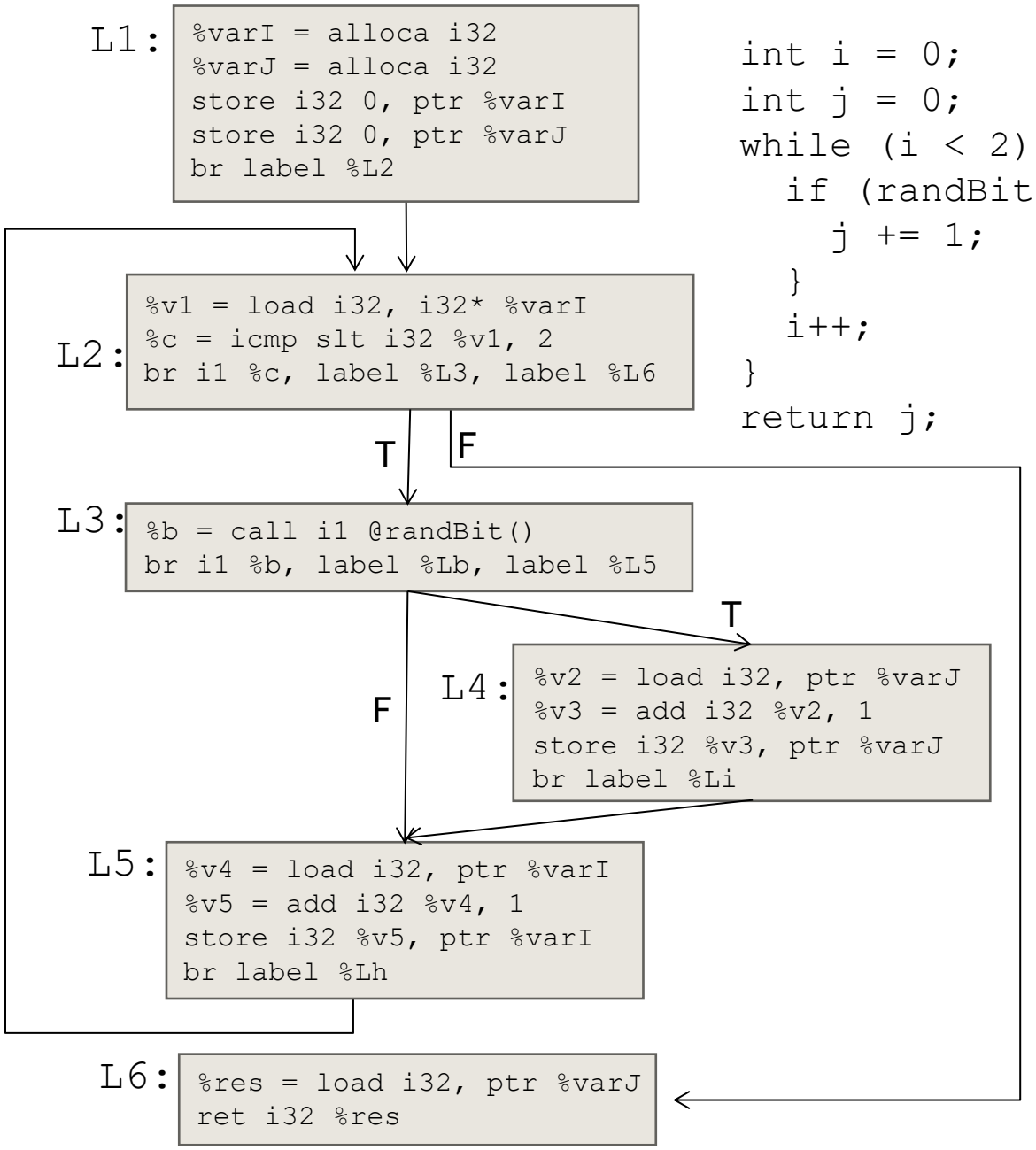
Perform an operation until it stops making progress

LECTURE OUTLINE

- Formalizing Dataflow
- Lattices
- Abstract Interpretation



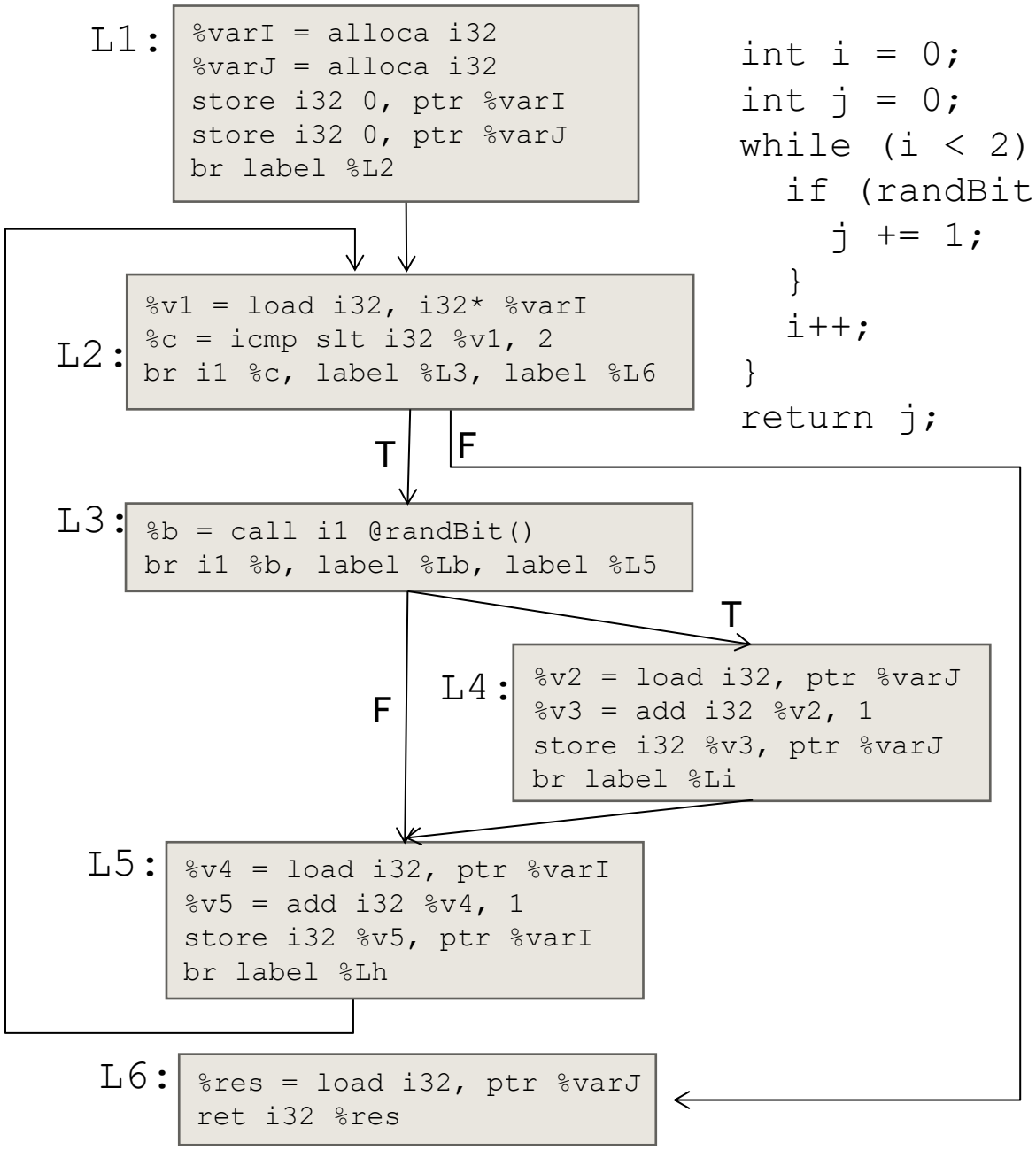
DATAFLOW SATURATION REVIEW




```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```

Idea 1: Allow the br instruction to partition value sets according to the condition

DATAFLOW SATURATION REVIEW




```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```

	*varI	*varJ	
L1 in	ANY	ANY	
L1->L2	ANY {0}	ANY {0}	
L2 in	ANY	ANY	U L1→L2, L5→L2
L2->L3	ANY {0,1}	ANY	
L2->L6	ANY {2,3,...}	ANY	
L3 in	ANY {0,1}	ANY	L2→L3
L3->L4	ANY {0,1}	ANY	
L3->L5	ANY	ANY	
L4 in	ANY {0,1}	ANY	L3→L4
L4->L5	ANY	ANY	
L5 in	ANY {0,1}	ANY	U L3→L5, L4→L5
L5->L2	ANY {0,1,2}	ANY	
L6 in	ANY {2,3,...}	ANY	L2→L6
L6 end	ANY {2,3,...}	ANY	

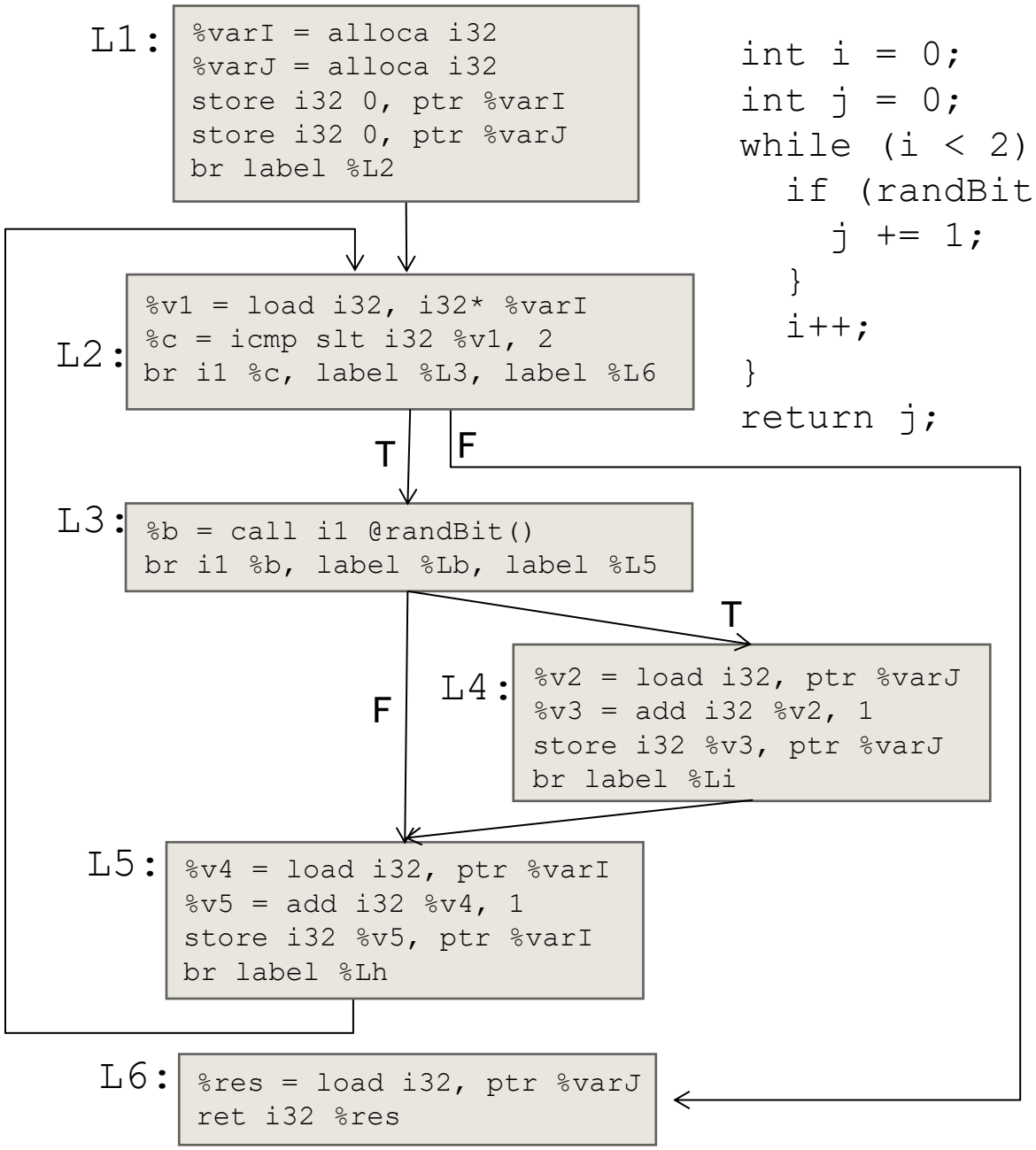
DATAFLOW SATURATION REVIEW


```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
rn j;
```

	*varI	*varJ	
L1 in	ANY		
L1->L2		ANY {0}	
	ANY	ANY	U L1→L2 L5→L2
L2->L3	{0,	ANY	
L2	ANY {2,3,	Y	
L3	ANY {0,1}	NY	3
L3->	{0,1}	AN	
->L5	ANY		
L4 in	{0,1}	ANY	L3→L4
->L5	ANY	ANY	
L5 in	ANY {0,1}	ANY	U L3→L5 L4→L5
L5->L2	ANY {0,1,2}	ANY	
L6 in	ANY {2,3,...}	ANY	L2→L6
L6 end	ANY {2,3,...}	ANY	

USELESS

DATAFLOW SATURATION REVIEW

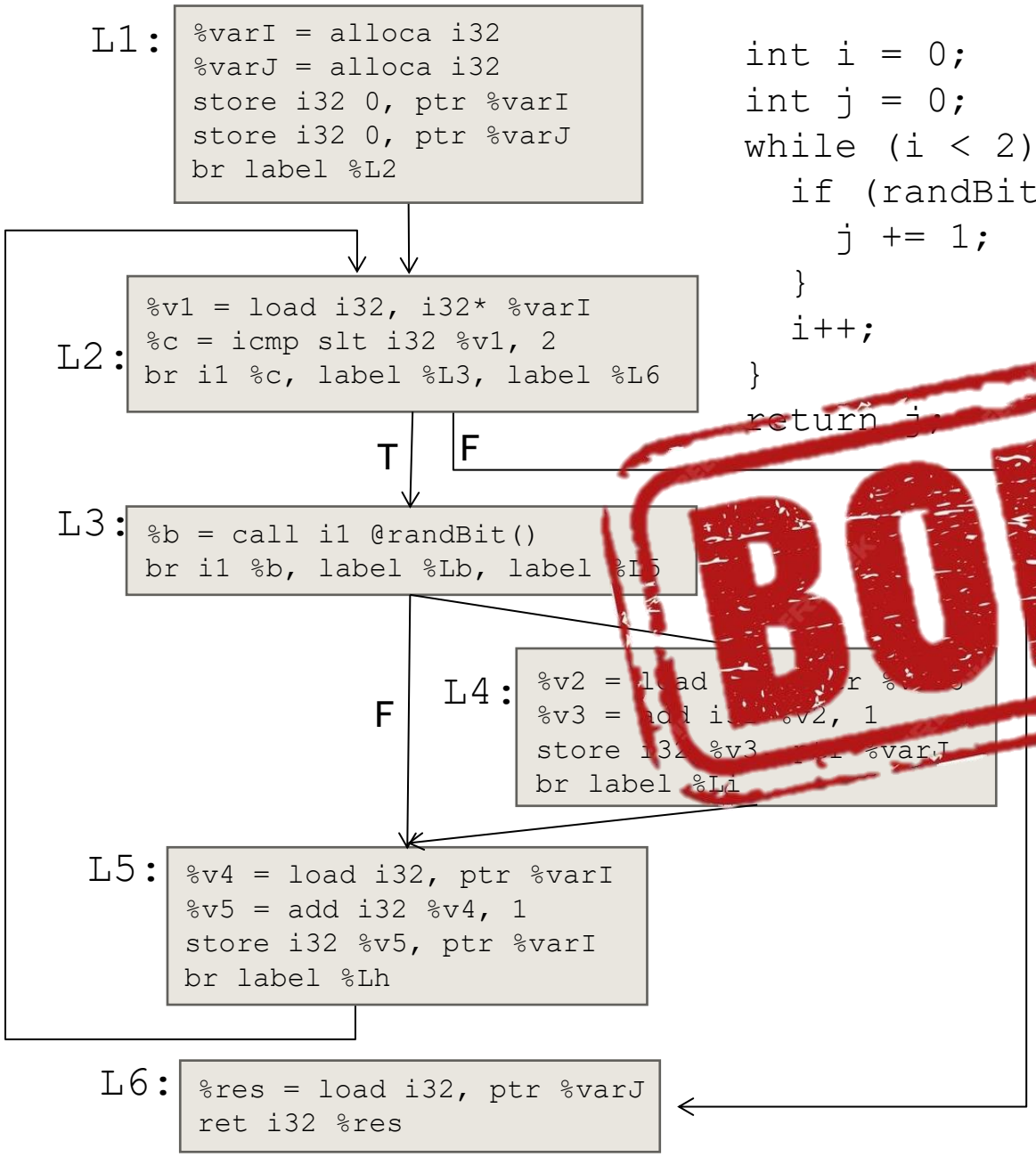



```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```

Idea 1: Allow the br instruction to partition value sets according to the condition

Idea 2: Create a distinguished uninitialized value (?) That is distinct from the ANY value

DATAFLOW SATURATION REVIEW




```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```

	*varI	*varJ	
L1 in	?	?	
L1->L2	? {0}	? {0}	
L2 in	? {0} {0,1} {0,1,2}	? {0} {0,1} {0,1,2}	U L1→L2 L5→L2
L2->L3	? {0} {0,1}	? {0} {0,1}	
L2->L6	{2}	? {0,1,2}	
L3 in	? {0} {0,1}	? {0} {0,1}	L2→L3
L3->Lb	? {0,1}	? {0} {0,1}	
L3->L5	? {0} {0,1}	? {0} {0,1}	
L4 in	? {0} {0,1}	? {0} {0,1}	L3→L4
L4->L5	? {0} {0,1}	? {1} {1,2}	
L5 in	? {0} {0,1}	? {0,1} {0,1,2}	U L3→L5 L4→L5
L5->L2	? {1} {1,2}	? {0,1} {0,1,2}	
L6 in	? {2}	? {0,1,2}	L2→L6
L6 end	? {2}	? {0,1,2}	

GOALS AND NEEDS

ABSTRACT INTERPRETATION

Goal: find potential divide-by-zero operations

Avert availability issue

Problem #1: value uncertainty

Problem #2: unbounded paths

Problem #3: cyclic dependency

Problem #4: tedious Saturation

SHORTCUT TO SATURATION

DATAFLOW FRAMEWORKS

OBSERVATION

You necessarily reach saturation once
your set contains EVERY possible value

WIDENING

Place extra values in the set

IMPLEMENTATION CONSIDERATIONS

DATAFLOW FRAMEWORKS

How would you code up a flow-sensitive DFA?

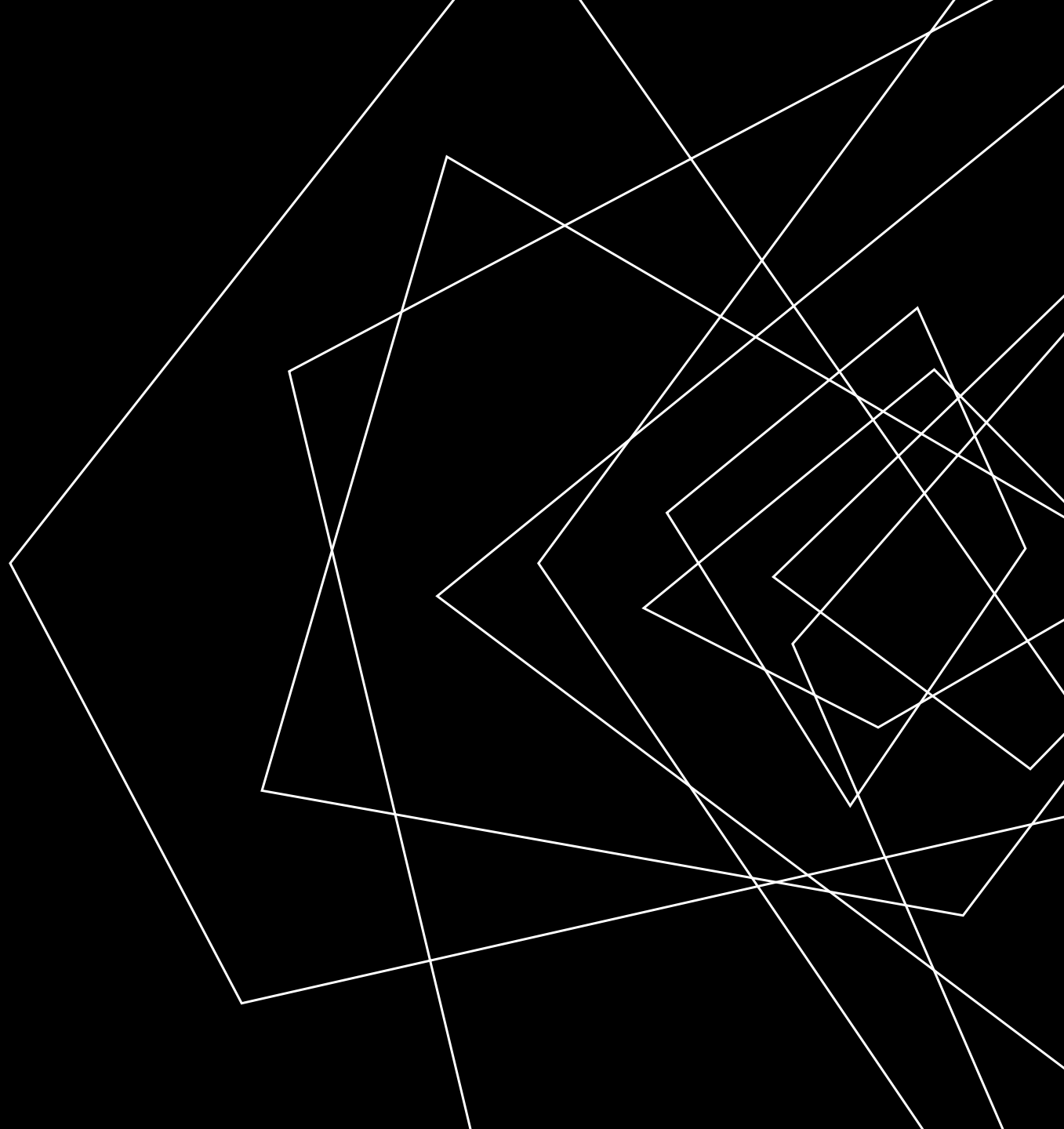
Efficient (and correct!) data representation

Efficient (guaranteed terminating!) runtime

Generic / re-usable

LECTURE OUTLINE

- Abstract Interpretation
- Generalizing Dataflow
- Dataflow Frameworks



ABSTRACT PLACEHOLDERS

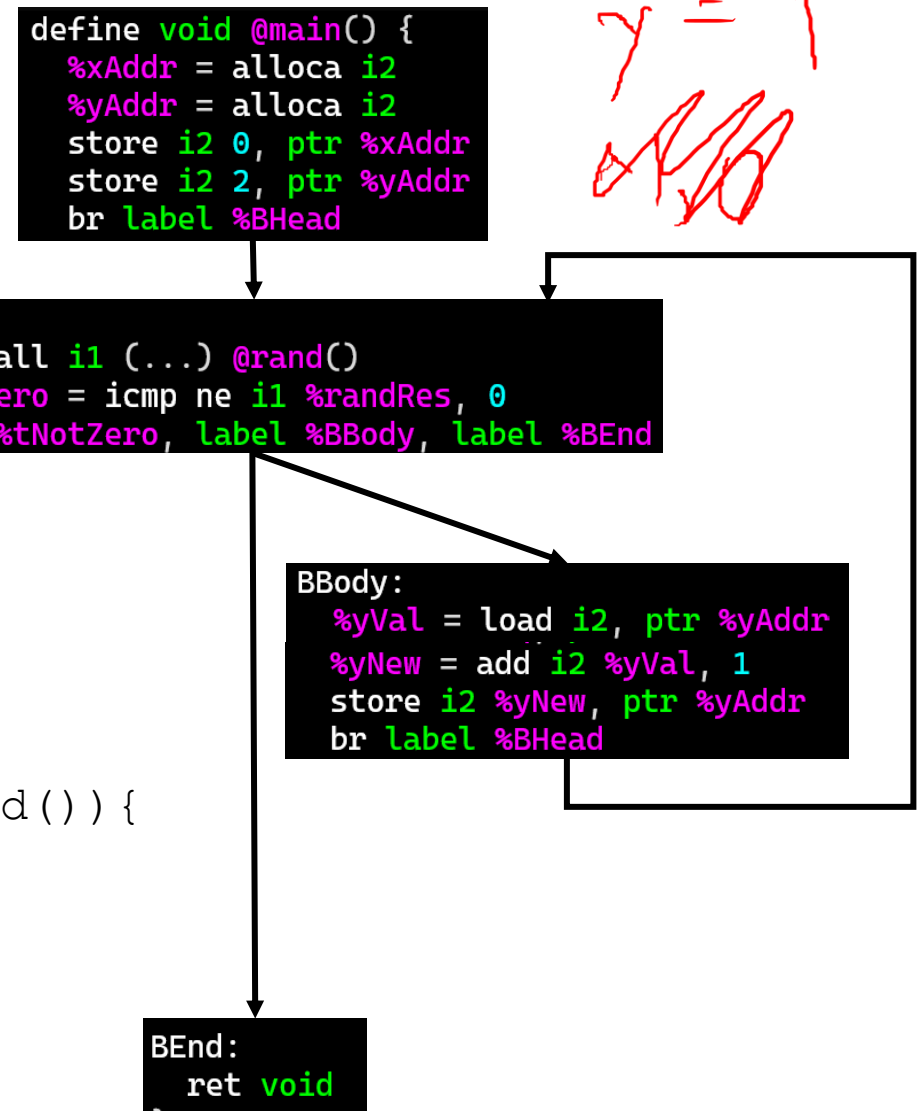
DATAFLOW FRAMEWORKS

ARE YOU REALLY GOING TO CARRY AROUND THE SET OF ALL INTEGERS?

- Use a placeholder for the “ALL INTEGERS” set

WORKS GREAT FOR LARGE SETS!

```
1. int2 x = 0;
2. int2 y = 2;
3. int1 t;
4. while (t=rand()){
5.   y = y+1;
6. }
7. return y;
```



ABSTRACT PLACEHOLDERS

DATAFLOW FRAMEWORKS

ARE YOU REALLY GOING TO CARRY AROUND THE SET OF ALL INTEGERS?

- Use a placeholder for the “ALL INTEGERS” set

WORKS GREAT FOR LARGE SETS!

- While we’re at it, maybe we can abstract other values-set too

```
1. int32 x = 0;
2. int32 y = 2;
3. int1 t;
4. while (t=rand()){
5.   y = y+1;
6. }
7. return y;
```

```
define i32 @main() {
  %xAddr = alloca i32
  %yAddr = alloca i32
  store i32 0, ptr %xAddr
  store i32 2, ptr %yAddr
  br label %BHead
```

```
BHead:
  %t = call i1 (...) @rand()
  %tNotZero = icmp ne i1 %randRes, 0
  br i1 %tNotZero, label %BBody, label %BEnd
```

```
BBody:
  %yVal = load i32, ptr %yAddr
  %yNew = add i32, 1, %yVal
  store i32 %yNew, ptr %yAddr
  br label %BHead
```

```
BEnd:
  %yRet = load i32, ptr %yAddr
  ret i32 %yRet
```

ABSTRACT DOMAINS

DATAFLOW FRAMEWORKS

BIG IDEA:

OPERATE OVER ABSTRACT VALUES THAT REPRESENT SETS OF CONCRETE VALUES

FROM THE CONCRETE DOMAIN TO THE ABSTRACT

$\alpha(\{0\})$ = **zero**

$\alpha(S)$ = if all values in S are greater than 0 then **pos**

else if all values in S are less than 0 then **neg**

else **num**

FROM THE ABSTRACT DOMAIN TO THE CONCRETE

γ(zero) = {0}

γ(pos) = {all positive ints}

γ(neg) = {all negative ints}

γ(num) = Int (i.e., all ints)

IS IT CORRECT?

DATAFLOW FRAMEWORKS

HOW WOULD YOU CODE UP A FLOW-SENSITIVE DFA?

Efficient (and correct!) data representation

Efficient (guaranteed terminating!) runtime

Generic / re-usable

Constraints needed:

- On “ranking” domain elements
- On combining domain elements

DOMAIN NEEDS

DATAFLOW FRAMEWORKS

SOME BASIC DEFINITIONS

A **partially-ordered set** (poset) is a set S and a partial ordering ⊆, such that the ordering ⊆ is:

- Reflexive
- Anti-symmetric
- Transitive

A **lattice** is a poset in which each pair of elements has

- A least upper bound (the *join*)

  for x and y, the join z is defined such that:

  - x ⊆ z and
  - y ⊆ z and
  - for all w such that x ⊆ w and y ⊆ w, w ⊇ z

No upper bound lower than z

z is actually an upper bound

- A greatest lower bound (the *meet*)

  basically the same deal, but reversed

A **complete lattice** is a lattice in which all subsets have a meet and join

Example 1: S: English words, ⊆ substring

Poset: ✓ Lattice: ✗

Example 2: S: English words, ⊆ shorter or equal in length

Poset: ✗ Lattice: ✗

Example 3: S: integers, ⊆ as lte

Poset: ✓ Lattice: ✓

Example 4: S: integers, ⊆ as lt

Poset: ✗ Lattice: ✗

Example 5: S: set of all sets of letters, ⊆ is subset

Poset: ✓ Lattice: ✓

FACT DATATYPE NEEDS

DATAFLOW FRAMEWORKS

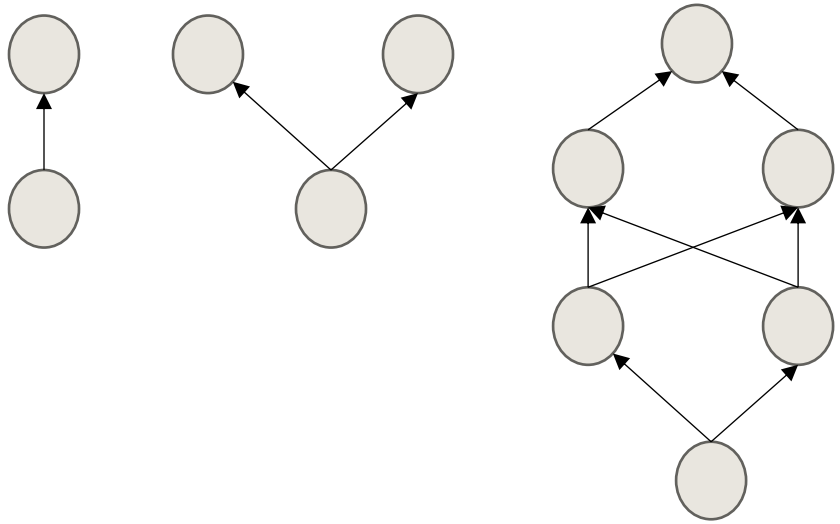
FORMAL(ISH) REQUIREMENTS

- A fact datatype (ideally of unbounded size)
- An ordering that indicates progress
- Unique solution
- A guarantee of a finite number of steps to hit the maximum value
- An update step that never loses progress

TRY-OUTS

- poset
- Lattice
- Complete lattice

POSET



FACT DATATYPE NEEDS

DATAFLOW FRAMEWORKS

FORMAL(ISH) REQUIREMENTS

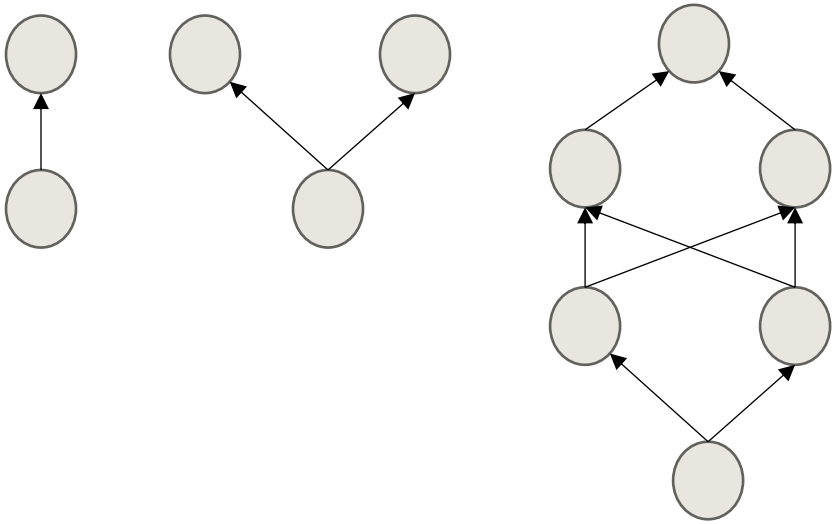
- A fact datatype (ideally of unbounded size)
- An ordering that indicates progress
- Unique solution
- A guarantee of a finite number of steps to hit the maximum value
- An update step that never loses progress

TRY-OUTS

- poset
- Lattice
- Complete lattice

LATTICE

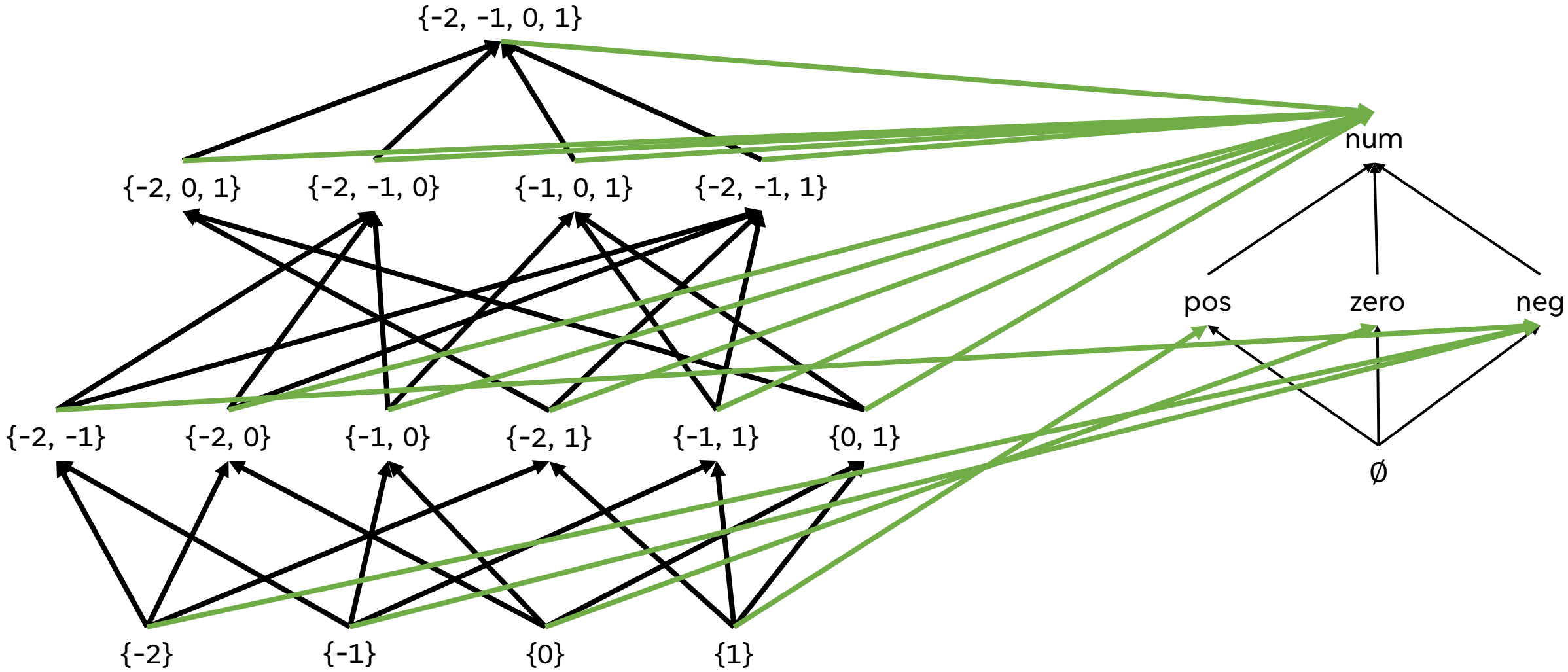
Poset with a least upper bound and a greatest lower bound



A VERY APPROXIMATE LATTICE

ABSTRACT INTERPRETATION

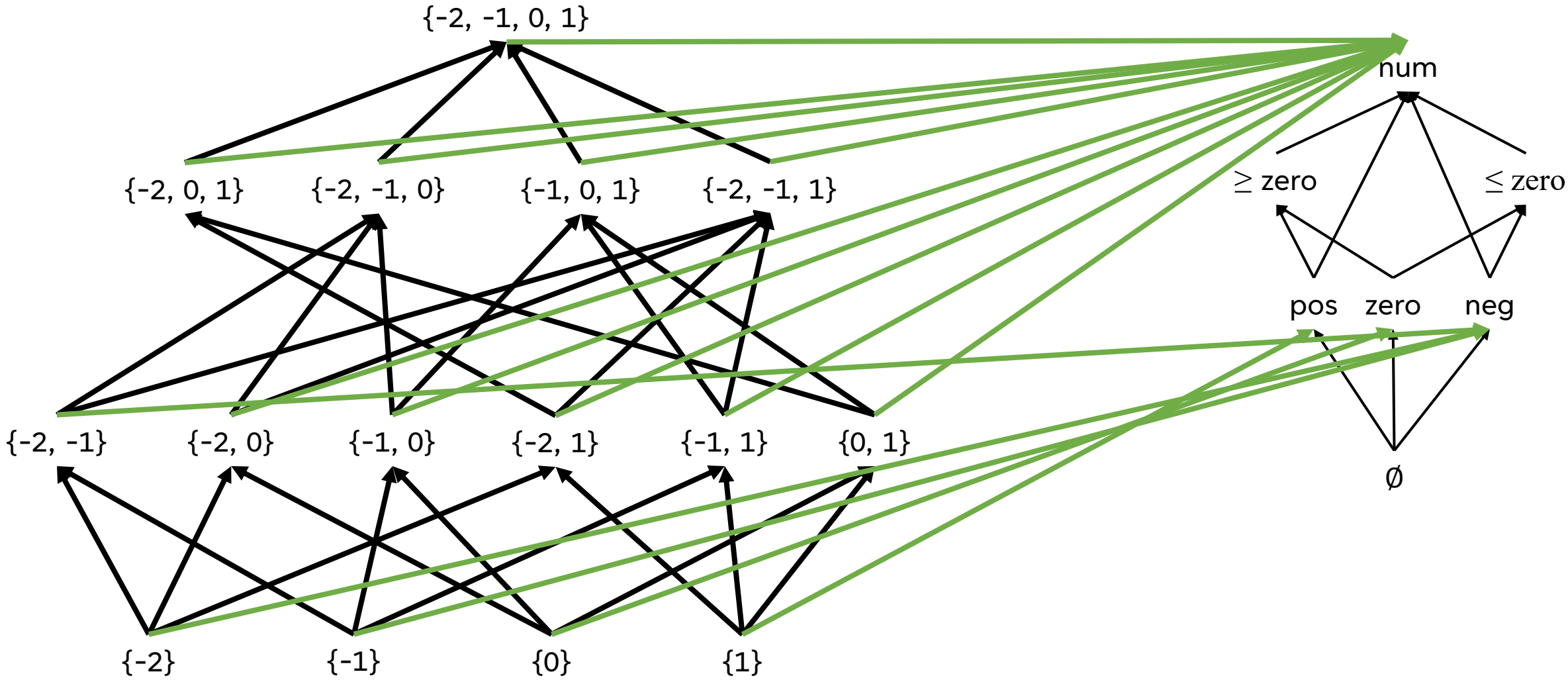
Abstract Domain of Signs



A LESS-APPROXIMATE LATTICE

ABSTRACT INTERPRETATION

ABSTRACT DOMAIN OF SIGNS



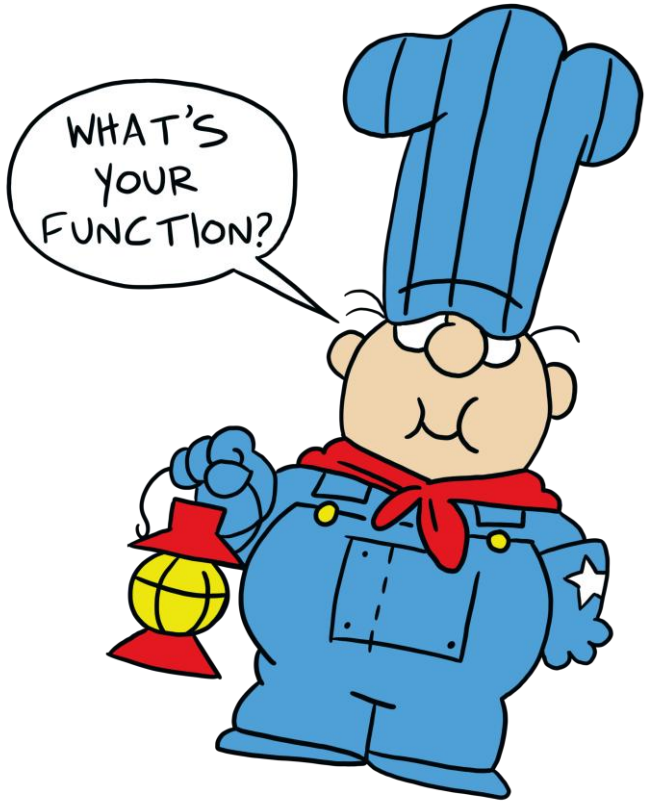
FUNCTION NEEDS

DATAFLOW FRAMEWORKS

Some Basic Definitions

A function f is a **monotonic function** if
$x \subseteq y$ implies $f(x) \subseteq f(y)$

An element z is a **fixpoint** of f iff $z = f(z)$



FUNCTION NEEDS

DATAFLOW FRAMEWORKS

SOME BASIC DEFINITIONS

A function f is a **monotonic function** if
x ⊆ y implies f(x) ⊆ f(y)

An element z is a **fixpoint** of f iff z = f(z)

Example

$f(x) = x \cup \{ b \}$

$f(\{b\}) = \{ b \}$ ← $\{b\}$ is a fixpoint of f

$f(\{a\}) = \{ a, b \}$ ← $\{a\}$ is **not** a fixpoint of f

$f(\{a, b\}) = \{ a, b \}$ ← $\{a, b\}$ is a fixpoint of f

$f(f(\{a\}))$ is a fixpoint of f

$f(f(any\ set))$ is a fixpoint of f

WHY DOES THIS MATTER?

DATAFLOW FRAMEWORKS

Every finite lattice
is a complete lattice

PRACTICAL UPSHOT

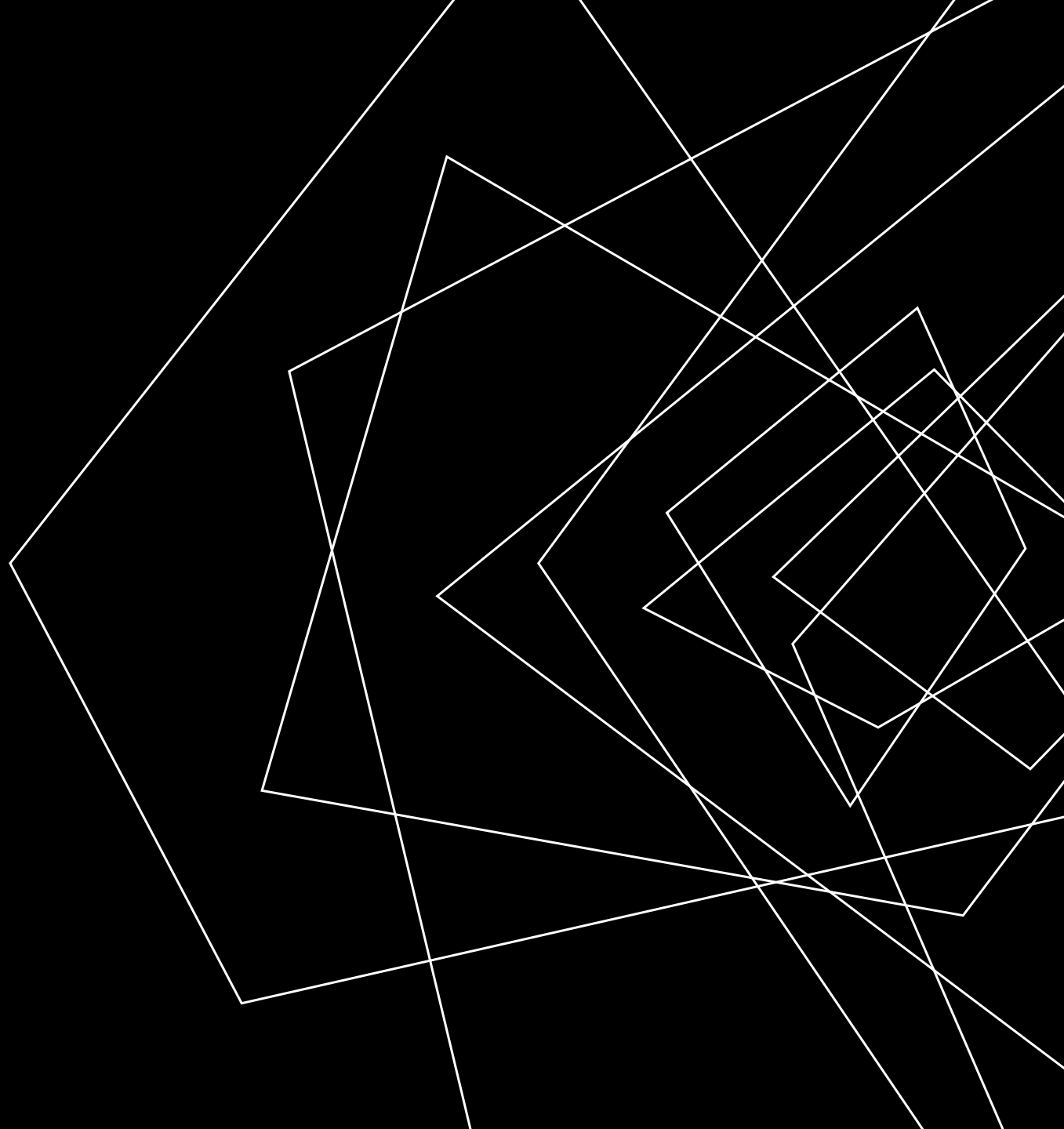
If L is a complete lattice and f is monotonic, then f has a greatest fixpoint and a least fixpoint

If L has no infinite ascending chains, the least fixpoint can be computed by iterative application of f

So the analysis will terminate

LECTURE OUTLINE

- Enhancing Dataflow analysis
- Lattices
- Abstract Interpretation



ANALYSIS PRECISION

ABSTRACT INTERPRETATION

PRECISION / EFFICIENCY TRADEOFF

With a complete lattice we can, in theory, eventually terminate

That's not a very strong guarantee!

The shallower the lattice, the faster the fixpoint

Choose to approximate the lattice

CHAOTIC ITERATION

STATIC ANALYSIS: CONTROL FLOW GRAPHS

A WORKLIST ALGORITHM

- Select the next worklist item in any order
- Necessarily assumes progress towards some goal

DEALING WITH "UNCOMPUTED" SETS

- Assume a reasonable "initial" value

Surprisingly, not a band with merch at Hot Topic

ANALYSIS PRECISION

ABSTRACT INTERPRETATION

PRECISION / EFFICIENCY TRADEOFF

With a complete lattice we can, in theory, eventually terminate

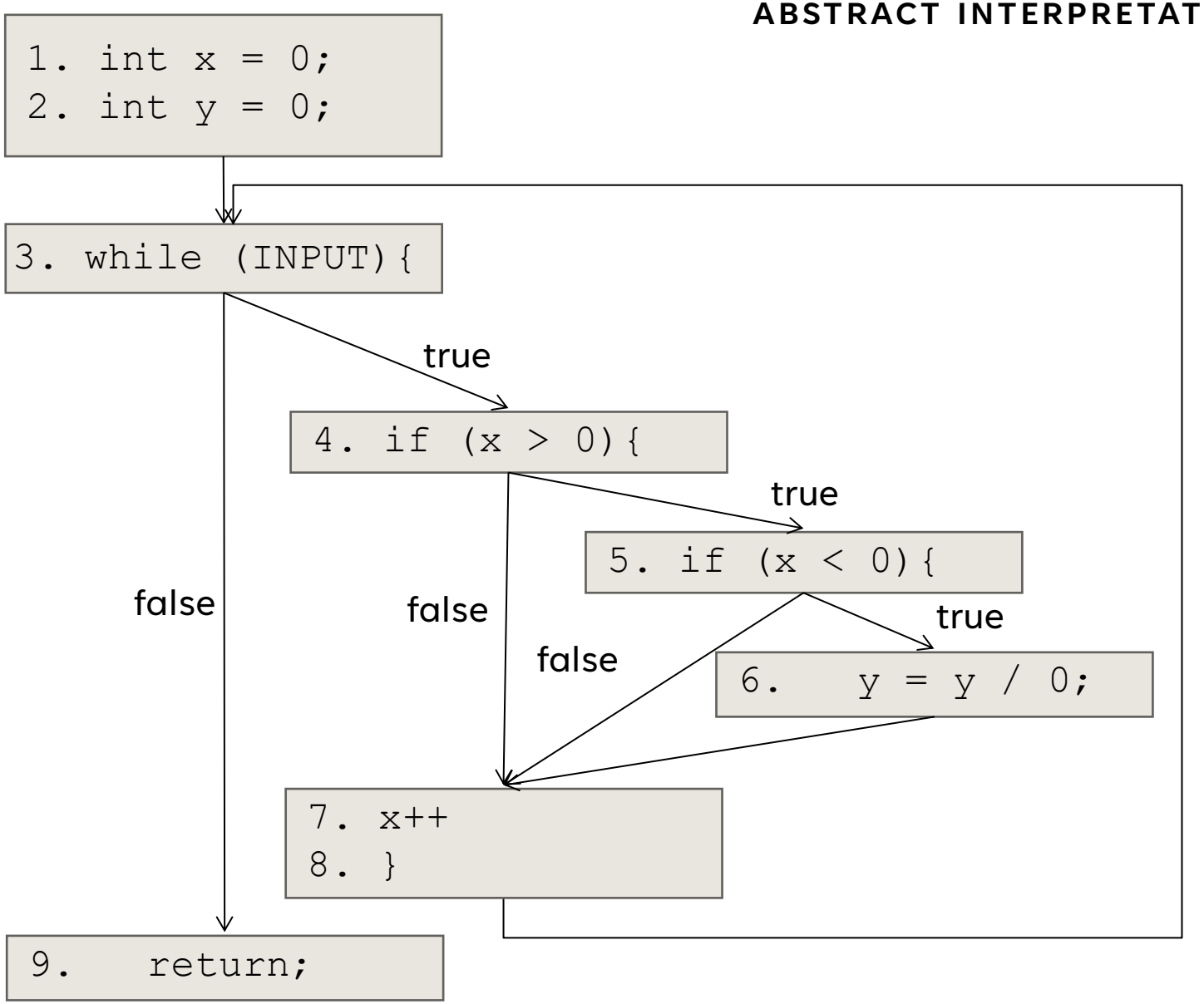
That's not a very strong guarantee!

The shallower the lattice, the faster the fixpoint

Choose to approximate the lattice

ANALYSIS PRECISION

ABSTRACT INTERPRETATION




```
1.  int x = 0;
2.  int y = 0;
3.  while (INPUT){
4.   if (x > 0)
5.     if (x < 0)
6.       y = y / 0;
7.   x++;
8.   }
9.   return;
```

ABSTRACT DOMAINS IN PRACTICE

STATIC ANALYSIS

"Singleton Integer Sets"

- You know the number, or you don't

Intervals

- You know a concrete range

Property Existence

- A property does or does not hold

SECTION SUMMARY
STATIC ANALYSIS

STATIC ANALYSIS GIVES US AN IMPORTANT GUARANTEE

- Completeness of bug finding / Soundness of verification
- Thus far we've been using source code

Anything that isn't crystal clear to a static analysis tool probably isn't clear to your fellow programmers, either. The classic hacker disdain for "bondage and discipline languages" is short-sighted – the needs of large, long-lived, multi-programmer projects are just different than the quick work you do for yourself.

- John Carmack's Static Code Analysis post