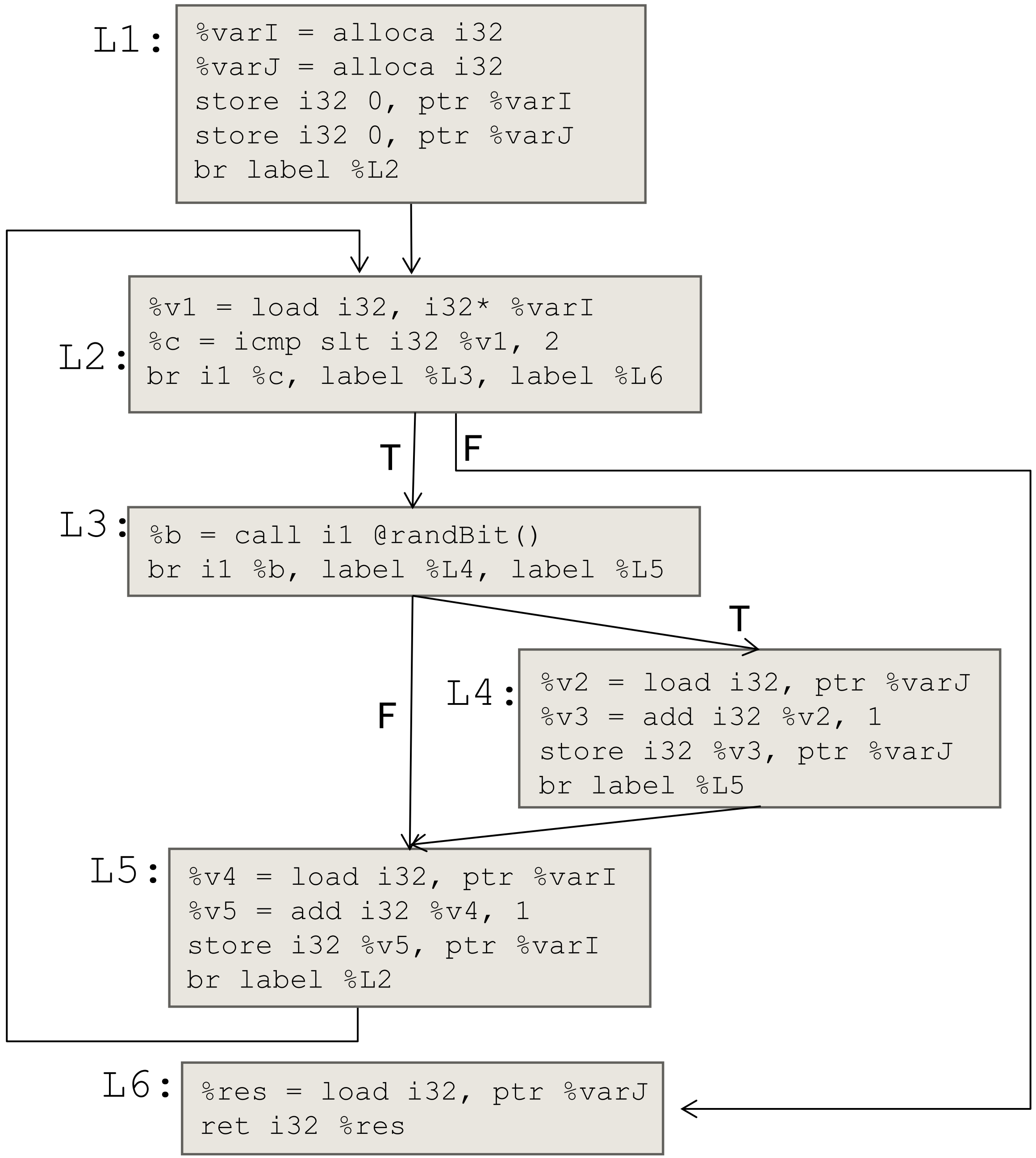
EXERCISE #11: DATAFLOW SATURATION REVIEW

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)

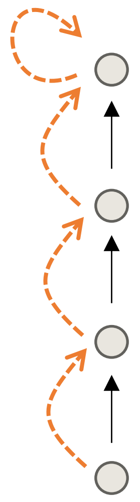
  Chaotic iteration

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

  Run the algorithm until it hits a fixpoint

Reaching Fixpoints Faster

- Intuitively: add some extra over-approximation

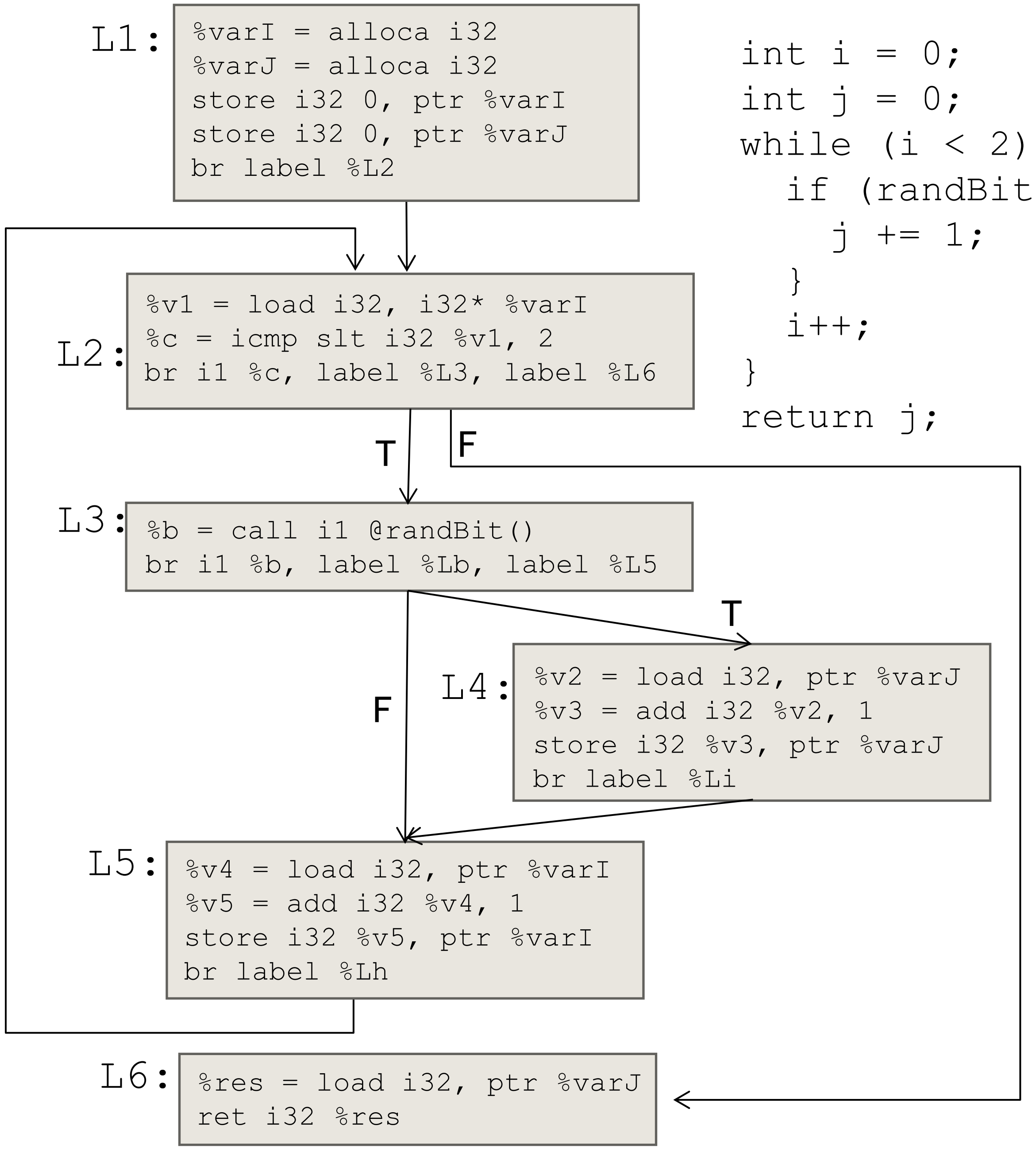


Perform an operation until it stops making progress

LECTURE OUTLINE

- Enhancing Dataflow analysis
- Lattices
- Abstract Interpretation

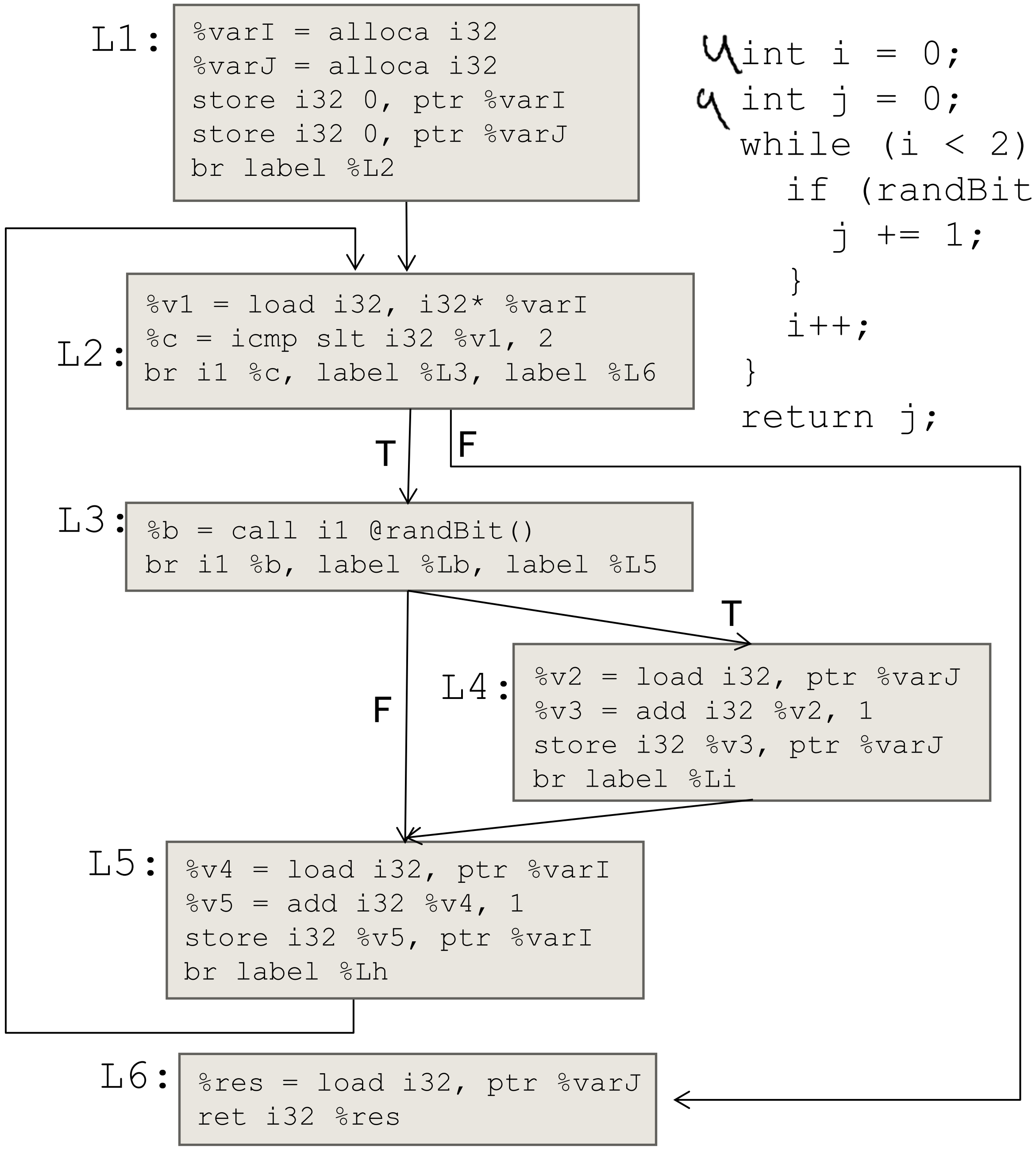
EXERCISE #7: DATAFLOW SATURATION REVIEW




```
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

EXERCISE #7: 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 {0,1}	ANY	
L4 in	ANY {0,1}	ANY	L3→L4
L4->L5	ANY {0,1}	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	

EXERCISE #7: DATAFLOW SATURATION REVIEW

USELESS

L1:
```
%varI = alloca i32
%varJ = alloca i32
store i32 0, ptr %varI
store i32 0, ptr %varJ
br label %L2
```

L2:
```
%v1 = load i32, i32* %varI
%c = icmp slt i32 %v1, 2
br i1 %c, label %L3, label %L6
```

T F

L3:
```
... randBit()
... %Lb, label %Lb, la...
```

T

L4:
```
... i32, ...
%v3 = ad...
... 3, ptr %varJ
b... %Li
```

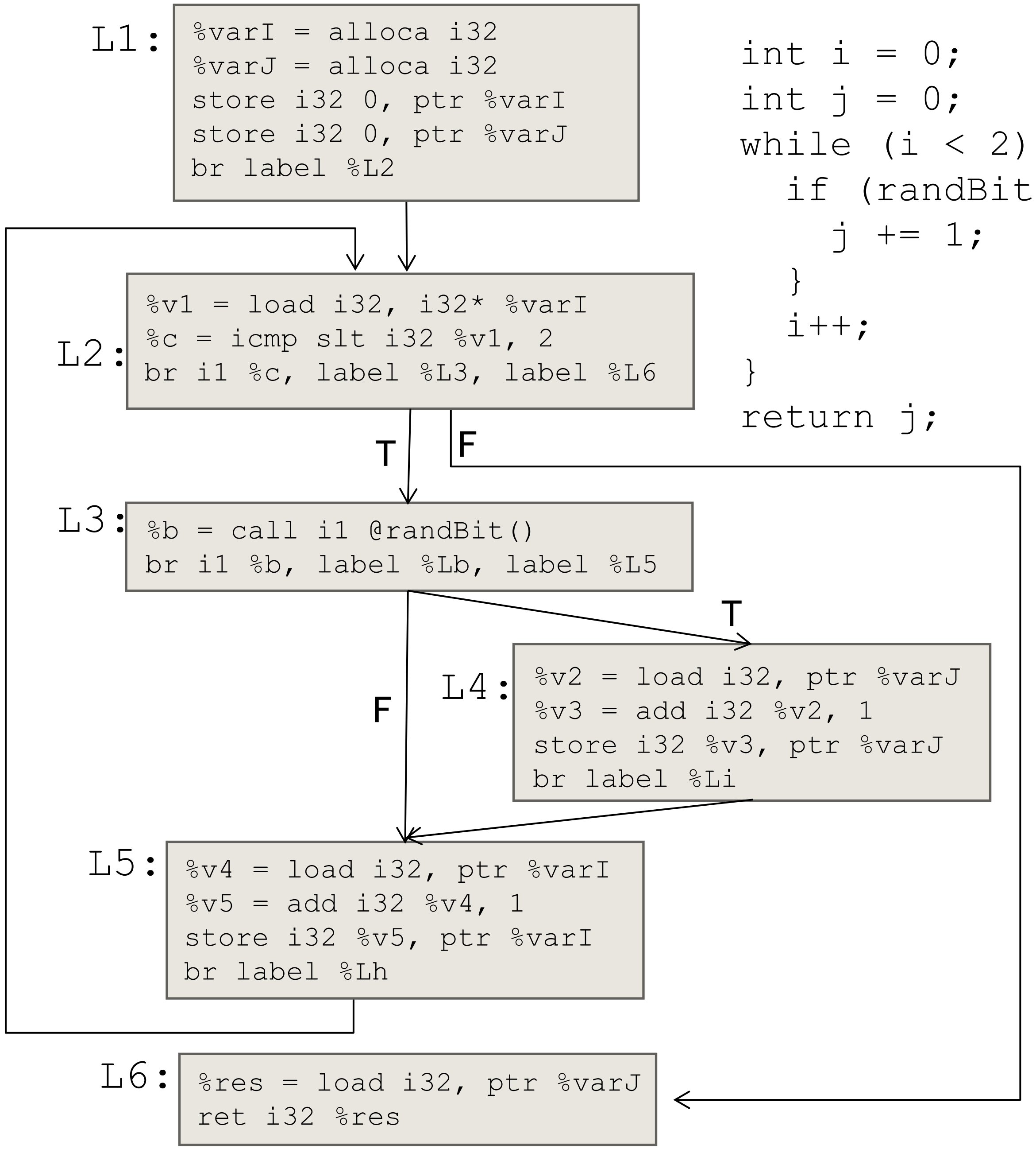
L5:
```
%v4 = ... i32, ptr %var...
%v5 = add i32 %v4...
store i32 ...
b...
```

L6:
```
%res = load i32, ptr %varJ
ret i32 %res
```

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

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

EXERCISE #7: DATAFLOW SATURATION REVIEW

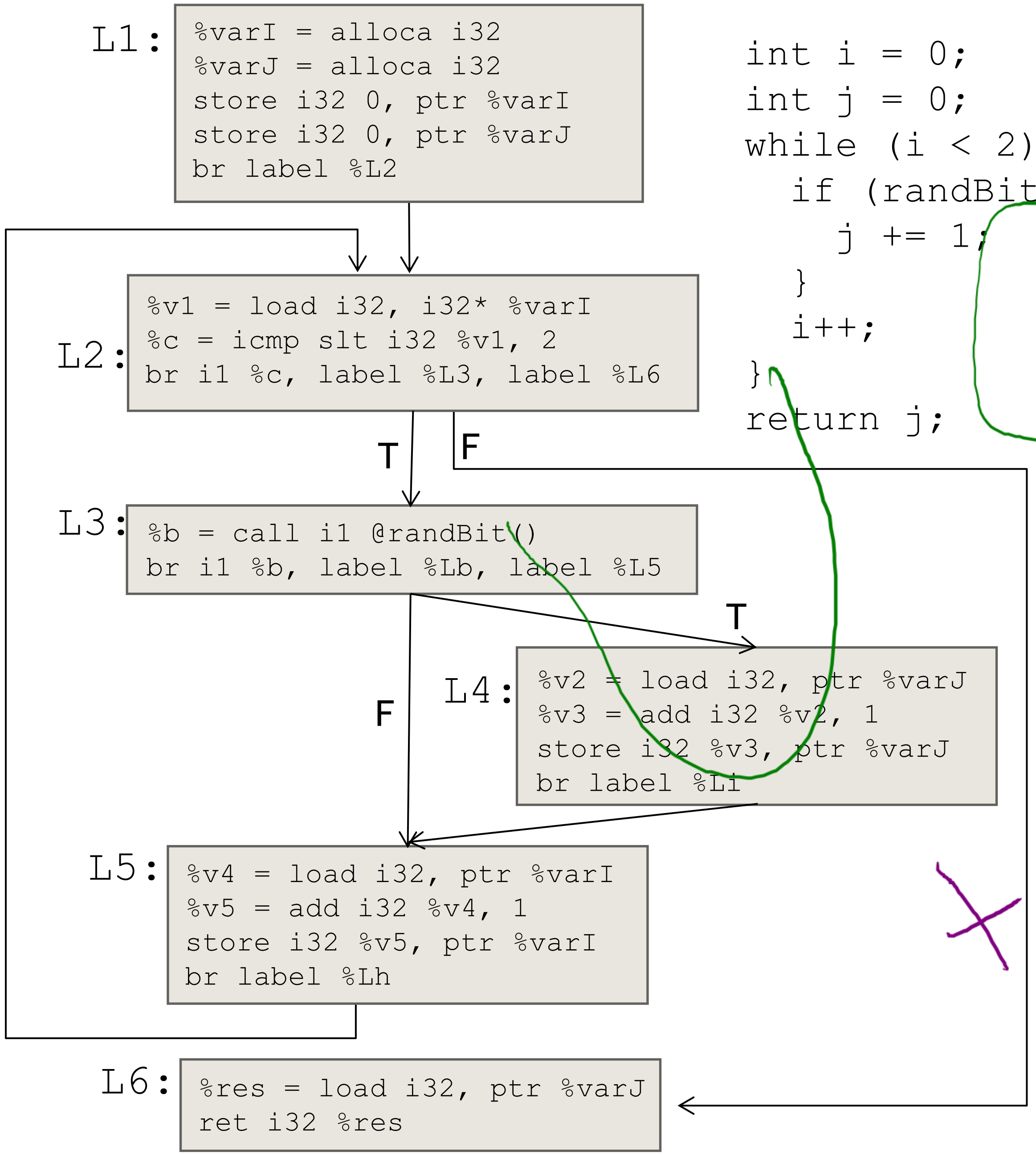



```
int i = 0;
int j = 0;
while (i < 2){
  if (randBit()){
    j += 1;
  }
  i++;
}
return j;
```

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

EXERCISE #7: 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->L4	? {0} {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}	

LECTURE OUTLINE

- Enhancing Dataflow analysis
- Lattices
- Abstract Interpretation

DOES OUR ANALYSIS TERMINATE?

ABSTRACT INTERPRETATION

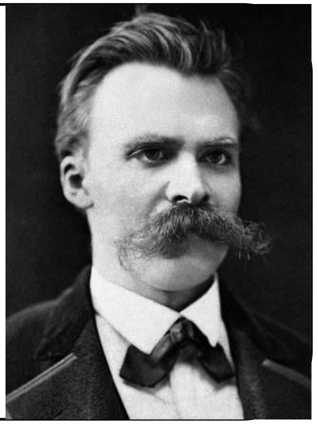
We've spent some time on some uncomfortable truths:

- Programs are often full of subtle bugs / vulnerabilities
- It is effectively impossible to tell if an arbitrary program terminates

Bad news: our analysis engine in a program

- What assurance do we have that our algorithm terminates as it analyzes an unknown / untrusted program?

> "Whoever fights monsters should see to it that in the process he does not become a monster. And when you look long into an abyss, the abyss also looks into you."
>
> \- Friedrich Nietzsche



FORMALIZING TERMINATION

DATAFLOW FRAMEWORKS

OUR VALUE-SET ANALYSES (APPEARED TO) HAVE SOME NICE PROPERTIES

- Guaranteed termination
- Completeness in values found

A COUPLE OF CONDITIONS HAPPENED TO OCCUR:

- A domain *D* of dataflow facts with a particular ordering
  Sets of possible integer values
- An operator to combine distinct dataflow facts
  Union over value-sets
- A dataflow function f_n: $D \rightarrow D$ that defines the effect of BBL_n
  Composition of the individual instruction transfer functions

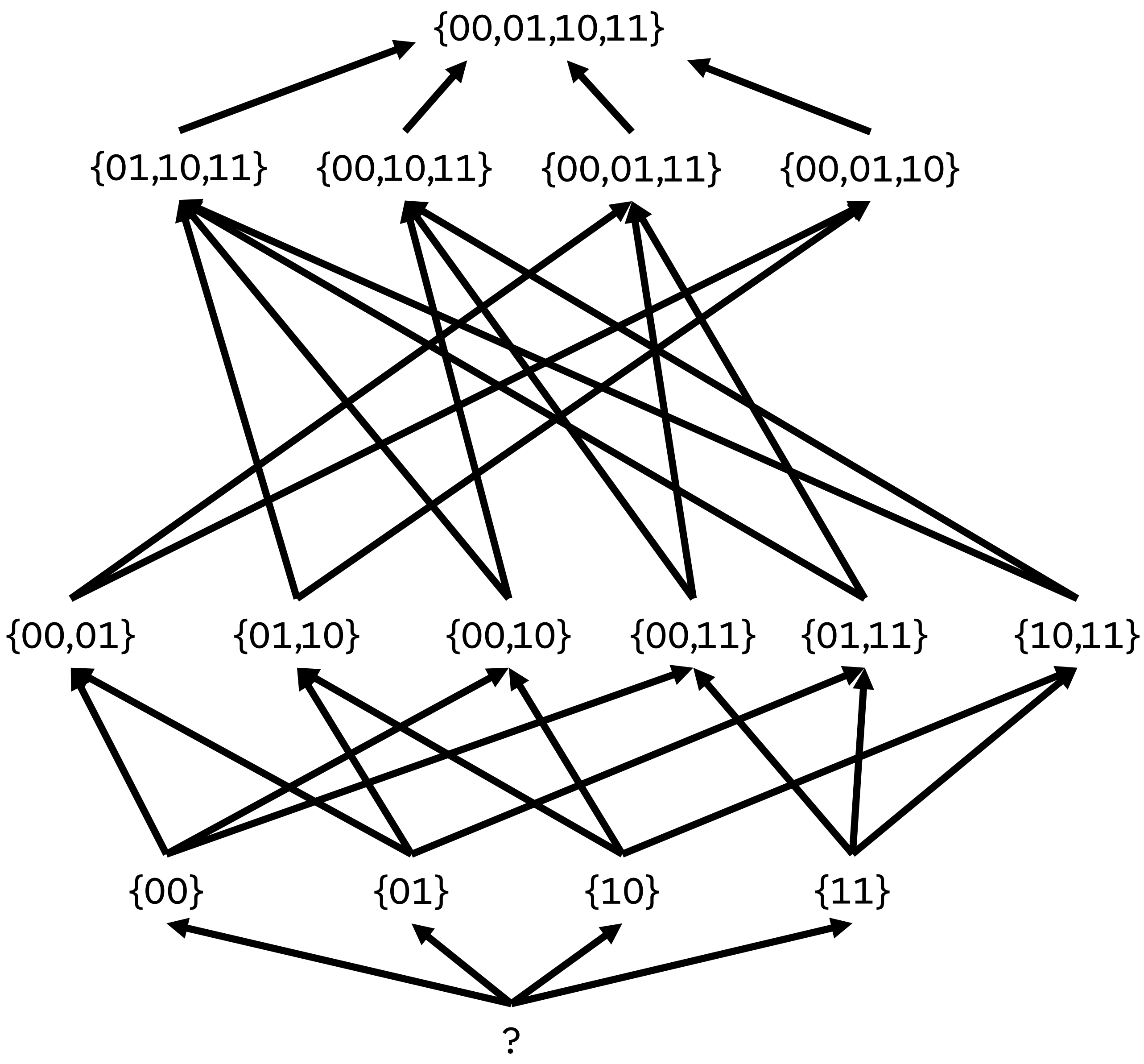
Claims

Bold Claims

FORMALIZING TERMINATION

DATAFLOW FRAMEWORKS

Value-Set "Rank"
(2-bit computer)



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

  z is actually an upper bound

  No upper bound lower than z

- 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: ✔