EECS 483: Compiler Construction

Lecture 19:
Dataflow Analysis

March 26
Winter Semester 2025

Announcements

- Exam grades still not quite done. Will review the grades and distribution on Monday
- March 27 Office Hours: Yuchen instead of Max
- Assignment 4 due next Friday. Get started!

CODE ANALYSIS

Assertion Removal

- Dynamic typing adds many runtime assertions into our program.

- ```
  let x = f() in
  let y = x + 2 in
  let z = y * x in
  ...
  ```
- Current compilation always adds assertions that inputs are integers

- ```
  x = f()
  assertInt(x)
  y = x + 2
  assertInt(y)
  assertInt(x)
  y2 = y >> 1
  z = y2 * x
  ...
  ```
- Which assertions can we remove?

Assertion Removal

- Dynamic typing adds many runtime assertions into our program.

- ```
  let x = f() in
  let y = x + 2 in
  let z = y * x in
  ...
  ```
- Current compilation always adds assertions that inputs are integers

- ```
  x = f()
  assertInt(x)
  y = x + 2
  assertInt(y)
  assertInt(x)
  y2 = y >> 1
  z = y2 * x
  ...
  ```
- **Which assertions can we remove?**

Assertion Removal

- When is it correct to remove an assertion from our SSA program?

  ```
  ...
  assertInt(x)
  ...
  ```

  When we are **sure** that the assertion will succeed

  In this case, if we are **sure** that x can only ever be a (tagged) integer at runtime.

- Appropriate analysis: determine what possible values x can take at runtime.

Possible Values Analysis

- To perform assertion removal, we need to figure out what possible values variables take at runtime.
  - Perform an analysis that says at every program point, the set of possible values that every variable might have at that point in the program.
  - Remove assertions that always would succeed on the possible values
- Rice's theorem applies: it's impossible to compute the exact correct sets. So we must approximate.

  Which way should we approximate?
  - Underapproximate: produce a **subset** of the true possible values. But might miss some
  - Overapproximate: produce a **superset** of the true possible values. But might include some that never happen
- For assertion removal, we need to **overapproximate**.
  - If our set is a superset of the true possible values, and still contains only tagged integers, then at runtime the possible value is definitely a tagged integer.
  - Might miss out on some assertion removals, but that's unavoidable.

Possible Values Analysis

- What do we mean by "possible values"?
- Performing this analysis at the SSA level. In SSA, a value is a 64-bit integer.
  - So we can represent a set of possible SSA values as a HashSet<i64> in Rust.

```
x = f()
assertInt(x)
y = x + 2
assertInt(y)
assertInt(x)
y2 = y >> 1
z = y2 * x
```

Problem: after x = f(), assuming f is an extern function, x may take on any value. That would be a huge set.
In general, a set of i64 values would take 2^(2^64) bits to represent!

Abstract Interpretation

- To keep space manageable, we need a different representation of sets, one that takes much less space than 2^(2^64) bits.
- This **inherently** means we are missing out on precision! But most of those sets are never going to come up in our analysis anyway.
- We design an "abstract domain" of possible value sets that is good enough to perform our analysis.
- To start, let's just worry about removing assertInt.
  - A simple abstract domain is to have just three elements:
    - Any (aka Top): this represents the set of all posssible 64-bit integers
    - Even (aka TaggedInt): this represents the even 64-bit integers
    - None aka Empty aka Bottom: the represents the empty set

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
x = y + z
```

What is the **most precise information** we know about the possible values of x based on the possible values of y and z?

Poss(x) = Flow[+](Poss(y), Poss(z)) ~~ { y + z | y in Poss(y), z in Poss(z) }
Flow[+](Any, Any) = Any
Flow[+](Any, Even) = Flow[+](Even, Any) = Any
Flow[+](Even, Even) = Even
Flow[+](None, Q) = None
Flow[+](P, None) = None

- Why is this correct? We output the most precise approximation of the set of all values that result from adding values in the input sets.
- None + Q = { y + z | y in EmptySet, z in Q } = EmptySet

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
x = y << n where n >= 1
```

Poss(x) = Flow[<< n](Poss(y)) ~~ { y << n | y in Poss(y) }
Flow[<< n](Any) = Even
Flow[<<n](Even) = Even
Flow[<<n](None) = None

- Note here that the case for Even **loses** precision:
  - { y << 1 | y in Even } = Multiples of 4 subset Even

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
x = y * z
```

What is the **most precise information** we know about the possible values of x based on the possible values of y and z?

Poss(x) = Flow[*](Poss(y), Poss(z)) ~~ { y * z | y in Poss(y), z in Poss(z) }
Flow[+](Any, Any) = Any
Flow[+](Any, Even) = Flow[+](Even, Any) = Even
Flow[+](Even, Even) = Even
Flow[+](None, Q) = None
Flow[+](P, None) = None

Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

```
assertInt(x)
```

Poss(x) = Flow[assertInt](Poss(x))
Flow[assertInt](Any) = Even
Flow[assertInt](Even) = Even
Flow[assertInt](None) = None

Straightline Code Example

```
0
 x = f()
1assertInt(x)
2y = x + 2
3assertInt(y)
4assertInt(x)
5y2 = y >> 1
6z = y2 * x
7
```

```
0:
1: {x: Any}
2: {x: Even}
3: {x: Even, y: Even}
4: {x: Even, y: Even}
5: {x: Even, y: Even}
6: {x: Even, y: Even, y2: Any}
7: {x: Even, y: Even, y2: Any, z: Any}
```

Straightline Code Example

```
0
x = f()
1assertInt(x)
2y = x + 2
3assertInt(y)
4assertInt(x)
5y2 = y >> 1
6z = y2 * x
7
```

```
0:
1: {x: Any}
2: {x: Even}
3: {x: Even, y: Even}
4: {x: Even, y: Even}
5: {x: Even, y: Even}
6: {x: Even, y: Even, y2: Any}
7: {x: Even, y: Even, y2: Any, z: Any}
```

Tag-checking Analysis

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, update that information accordingly
  To do a complete analysis: extend this to all SSA operations

- What about **blocks** and **functions?**

```
f(x):
  assertInt(x)
  assertInt(y)
  ...
```

What info do we have about x? about y?
Collect the info from all the places that **branch to f,** taking a "union"
We call these the **predecessors of f**, because they are the incoming edges of the control-flow graph.
Because we can have loops, **f** can be a predecessor of itself, so we have a similar circularity that we did in liveness.
Same solution: initialize the information to be minimal (bottom in this case) and update iteratively
For functions, the predeccessors are places that **call f**.
For main, there is a special implicit predecessor which is the entry point. This sets the input variable to Any because the program input is an array.

Loop Example

```
extern g
def main(y):
  def loop(i,a):
    if i == 0:
      a
    else:
      loop(i - 1, a + g())
  in
  loop(y, 0)
```

```
main(y):
  loop(i,a):
    thn():
      ret a
    els():
      assertInt(i)
      i' = i - 2
      x = g()
      assertInt(a)
      assertInt(x)
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

```
main(y):
  loop(i,a):
    thn():
     3 ret a
    els():
     4 assertInt(i)
     5 i' = i - 2
     6 x = g()
     7 assertInt(a)
     8 assertInt(x)
     9 a' = a + x
    10 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
  0 br loop(y, 0)
```

Initialize the blocks: main entry point arguments are Any, other blocks everything is None.

```
0: {y:Any}
1: {y,i,a:None}
2:
3: {y,i,a:None}
4: {y,i,a:None}
5:
6:
7:
8:
9:
10:
```

```
main(y):
  loop(i,a):
    thn():
    3
      ret a
    els():
    4 assertInt(i)
    5 i' = i - 2
    6 x = g()
    7 assertInt(a)
    8
      assertInt(x)
    9
      a' = a + x
   10
      br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0
  br loop(y, 0)
```

Next: apply the flow functions to update the internal nodes

```
0: {y:Any}
1: {y,i,a:None}
2: {y,i,a,b:None}
3: {y,i,a:None}
4: {y,i,a:None}
5: {y,i,a:None}
6: {y,i,a,i':None}
7: {y,i,a,i':None,x:Any}
8: {y,i,a,i':None,x:Even}
9: {y,i,a,i',a':None,x:Even}
10: {y,i,a,i',a':None,x:Even}
```

```
main(y):
  loop(i,a):
    thn():
     3
      ret a
    els():
     4assertInt(i)
     5i' = i - 2
     6x = g()
     7assertInt(a)
     8assertInt(x)
     9a' = a + x
    10br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
 0
  br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round	Next Round
0: {y:Any}	0: ?
1: {y,i,a:None}	1: ?
2: {y,i,a,b:None}	2:
3: {y,i,a:None}	3: ?
4: {y,i,a:None}	4: ?
5: {y,i,a:None}	5:
6: {y,i,a,i':None}	6:
7: {y,i,a,i':None,x:Any}	7:
8: {y,i,a,i':None,x:Even}	8:
9: {y,i,a,i',a':None,x:Even}	9:
10: {y,i,a,i',a':None,x:Even}	10:

```
main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
    1 b = i == 0
    2 cbr b thn() els()
  0 br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

	Previous Round	Next Round
0:	{y:Any}	{y:Any}
1:	{y,i,a:None}	{y:Any U None, i:Any U None, a: Even U None}
2:	{y,i,a,b:None}	
3:	{y,i,a:None}	?
4:	{y,i,a:None}	?
5:	{y,i,a:None}	
6:	{y,i,a,i':None}	
7:	{y,i,a,i':None,x:Any}	
8:	{y,i,a,i':None,x:Even}	
9:	{y,i,a,i',a':None,x:Even}	
10:	{y,i,a,i',a':None,x:Even}	

the loop(i,a) body 1 has two predecessors:

- br loop(y, 0)
- br loop(i', a')

Take the "union" of their information

```
main(y):
  loop(i,a):
    thn():
     3
     ret a
    els():
     4assertInt(i)
     5i' = i - 2
     6x = g()
     7assertInt(a)
     8
     assertInt(x)
     9a' = a + x
    10br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
 0
 br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

	Previous Round		Next Round
0:	{y:Any}	0:	{y:Any}
1:	{y,i,a:None}	1:	{y:Any,i:Any,a:Even}
2:	{y,i,a,b:None}	2:	
3:	{y,i,a:None}	3:	?
4:	{y,i,a:None}	4:	?
5:	{y,i,a:None}	5:	
6:	{y,i,a,i':None}	6:	
7:	{y,i,a,i':None,x:Any}	7:	
8:	{y,i,a,i':None,x:Even}	8:	
9:	{y,i,a,i',a':None,x:Even}	9:	
10:	{y,i,a,i',a':None,x:Even}	10:	

the loop(i,a) body 1 has two predecessors:

br loop(y, 0)

br loop(i', a')

Take the "union" of their information

```
main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
    1 b = i == 0
    2 cbr b thn() els()
  0 br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

	Previous Round	Next Round
0:	{y:Any}	{y:Any}
1:	{y,i,a:None}	{y:Any,i:Any,a:Even}
2:	{y,i,a,b:None}	
3:	{y,i,a:None}	{y,i,a,b:None}
4:	{y,i,a:None}	{y,i,a,b:None}
5:	{y,i,a:None}	
6:	{y,i,a,i':None}	
7:	{y,i,a,i':None,x:Any}	
8:	{y,i,a,i':None,x:Even}	
9:	{y,i,a,i',a':None,x:Even}	
10:	{y,i,a,i',a':None,x:Even}	

thn() and els() each have one predecessor
cbr b thn() els()

```
main(y):
  loop(i,a):
    thn():
     3 ret a
    els():
     4 assertInt(i)
     5 i' = i - 2
     6 x = g()
     7 assertInt(a)
     8 assertInt(x)
     9 a' = a + x
    10 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y,i,a,b:None}
4: {y,i,a,b:None}
5:
6:
7:
8:
9:
10:
```

```
main(y):
  loop(i,a):
    thn():
     3 ret a
    els():
     4 assertInt(i)
     5 i' = i - 2
     6 x = g()
     7 assertInt(a)
     8 assertInt(x)
     9 a' = a + x
    10 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

Update the rest of the internal nodes

Current Round

```
 0: {y:Any}
 1: {y:Any,i:Any,a:Even}
 2: {y:Any,i:Any,a:Even,b:Any}
 3: {y,i,a,b:None}
 4: {y,i,a,b:None}
 5: {y,i,a:None}
 6: {y,i,a,i':None}
 7: {y,i,a,i':None,x:Any}
 8: {y,i,a,i':None,x:Even}
 9: {y,i,a,i',a':None,x:Even}
10: {y,i,a,i',a':None,x:Even}
```

Since the results changed, we perform another iteration

```
main(y):
  loop(i,a):
    thn():
     3
      ret a
    els():
     4assertInt(i)
     5i' = i - 2
     6x = g()
     7assertInt(a)
     8
      assertInt(x)
     9a' = a + x
    10br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
 0
  br loop(y, 0)
```

	Previous Round		Next Round
0:	{y:Any}	0:	{y:Any}
1:	{y:Any,i:Any,a:Even}	1:	{y:Any,i:Any,a:Even}
2:	{y:Any,i:Any,a:Even,b:Any}	2:	
3:	{y,i,a,b:None}	3:	?
4:	{y,i,a,b:None}	4:	?
5:	{y,i,a:None}	5:	
6:	{y,i,a,i':None}	6:	
7:	{y,i,a,i':None,x:Any}	7:	
8:	{y,i,a,i':None,x:Even}	8:	
9:	{y,i,a,i',a':None,x:Even}	9:	
10:	{y,i,a,i',a':None,x:Even}	10:	

the loop(i,a) body 1 has two predecessors:
- br loop(y, 0)
- br loop(i', a')

Take the "union" of their information

```
main(y):
  loop(i,a):
    thn():
   3  ret a
    els():
   4  assertInt(i)
   5  i' = i - 2
   6  x = g()
   7  assertInt(a)
   8  assertInt(x)
   9  a' = a + x
  10  br loop(i', a')
 1  b = i == 0
 2  cbr b thn() els()
0 br loop(y, 0)
```

	Previous Round	Next Round
0:	{y:Any}	{y:Any}
1:	{y:Any,i:Any,a:Even}	{y:Any,i:Any,a:Even}
2:	{y:Any,i:Any,a:Even,b:Any}	
3:	{y,i,a,b:None}	{y:Any,i:Any,a:Even,b:Any
4:	{y,i,a,b:None}	{y:Any,i:Any,a:Even,b:Any
5:	{y,i,a:None}	
6:	{y,i,a,i':None}	
7:	{y,i,a,i':None,x:Any}	
8:	{y,i,a,i':None,x:Even}	
9:	{y,i,a,i',a':None,x:Even}	
10:	{y,i,a,i',a':None,x:Even}	

thn() and els() each have one predecessor
cbr b thn() els()

```
main(y):
  loop(i,a):
    thn():
     3 ret a
    els():
     4 assertInt(i)
     5 i' = i - 2
     6 x = g()
     7 assertInt(a)
     8 assertInt(x)
     9 a' = a + x
    10 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5:
6:
7:
8:
9:
10:
```

Update the rest of the internal nodes

```
main(y):
  loop(i,a):
    thn():
     3
     ret a
    els():
     4assertInt(i)
     5i' = i - 2
     6x = g()
     7assertInt(a)
     8
     assertInt(x)
     9a' = a + x
    10br loop(i', a')
   1b = i == 0
   2cbr b thn() els()
 0
 br loop(y, 0)
```

Current Round

```
 0: {y:Any}
 1: {y:Any,i:Any,a:Even}
 2: {y:Any,i:Any,a:Even,b:Any}
 3: {y:Any,i:Any,a:Even,b:Any}
 4: {y:Any,i:Any,a:Even,b:Any}
 5: {y:Any,i:Even,a:Even,b:Any}
 6: {y:Any,i:Even,a:Even,b:Any,i':Even}
 7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
 8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
 9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}
```

If we repeat one more iteration, we get the same result, and we have our final analysis

```
main(y):
  loop(i,a):
    thn():
     3 ret a
    els():
     4 assertInt(i)
     5 i' = i - 2
     6 x = g()
     7 assertInt(a)
     8 assertInt(x)
     9 a' = a + x
    10 br loop(i', a')
   1 b = i == 0
   2 cbr b thn() els()
 0 br loop(y, 0)
```

Update the rest of the internal nodes

Current Round

```
0: {y:Any}
1: {y:Any,i:Any,a:Even}
2: {y:Any,i:Any,a:Even,b:Any}
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5: {y:Any,i:Even,a:Even,b:Any}
6: {y:Any,i:Even,a:Even,b:Any,i':Even}
7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}
```

With all of this work, we can remove 1 assertion: assertInt(a)

We can't prove that assertInt(i) succeeds because the initial value of y might not be an integer...

```
extern g
def main(y):
  def loop(i,a):
    if i == 0:
      a - z
    else:
      loop(i - 1, a + g())
  in
  loop(y, 0)
```

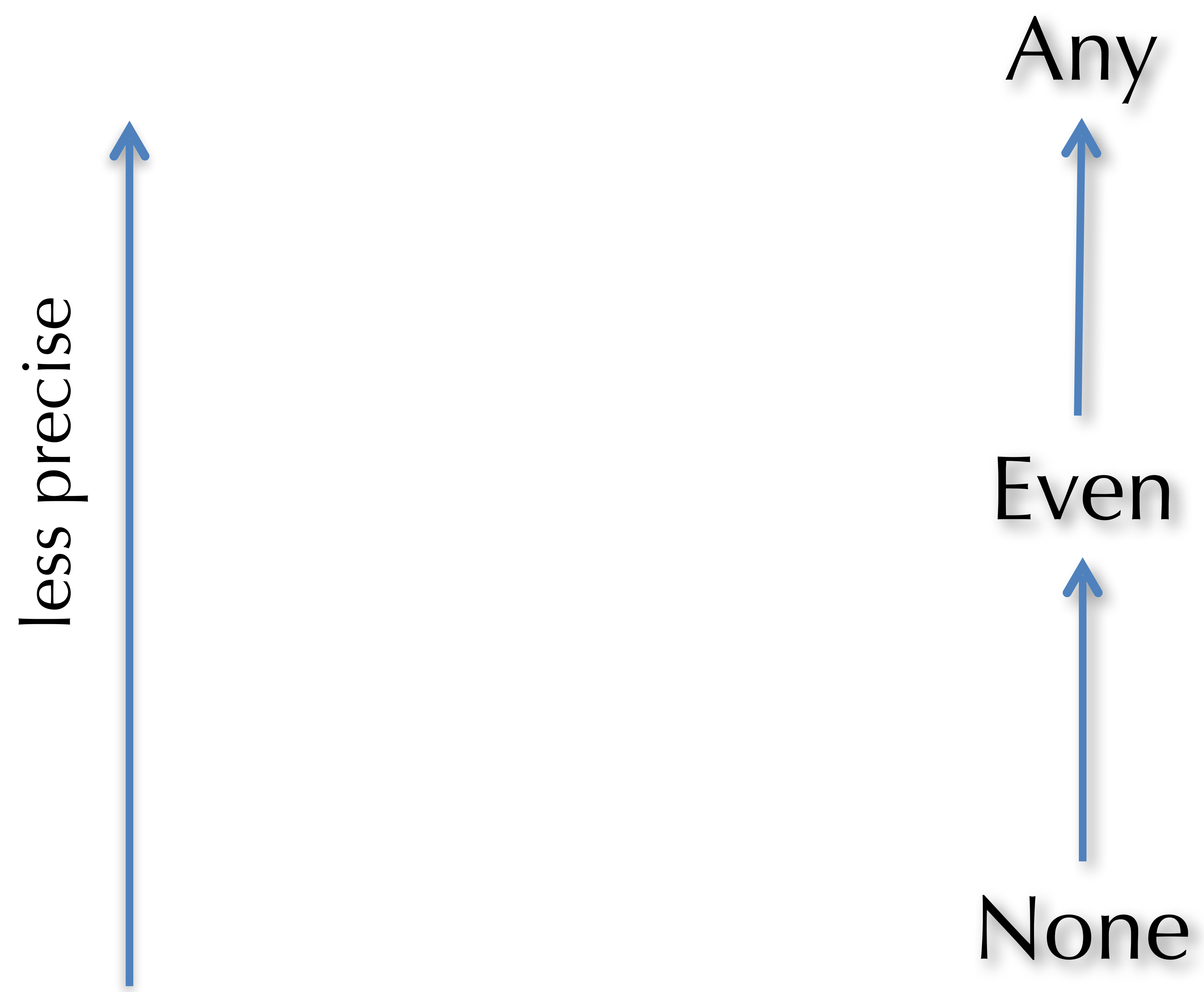
inline once

```
extern g
def main(y):
  def loop(i,a):
    if i == 0:
      a
    else:
      loop(i - 1, a + g())
  in
  if y == 0:
   a
  else:
    loop(y - 1, 0 + g())
```

If we re-do the analysis, no i is always an Int

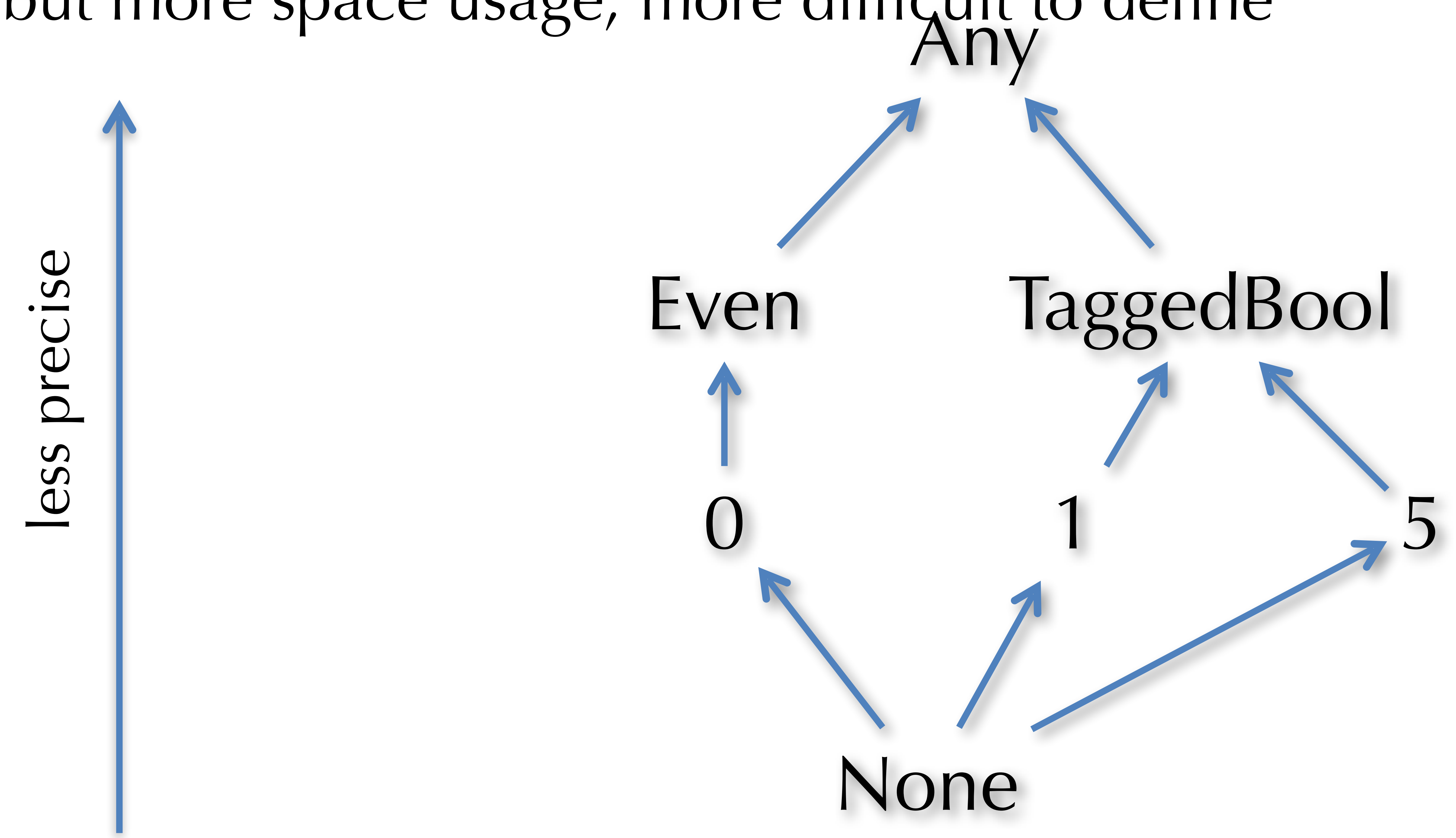
Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - A simple abstract domain is to have just three elements:
    - Any (aka Top): this represents the set of all posssible 64-bit integers
    - Even (aka TaggedInt): this represents the even 64-bit integers
    - None aka Empty aka Bottom: the represents the empty set



Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - What about for Booleans?
  - Update all flow functions accordingly
  - Similar for arrays
  - Tradeoff: more complex Abstraction means more precise analysis, but more space usage, more difficult to define



GENERAL DATAFLOW ANALYSIS

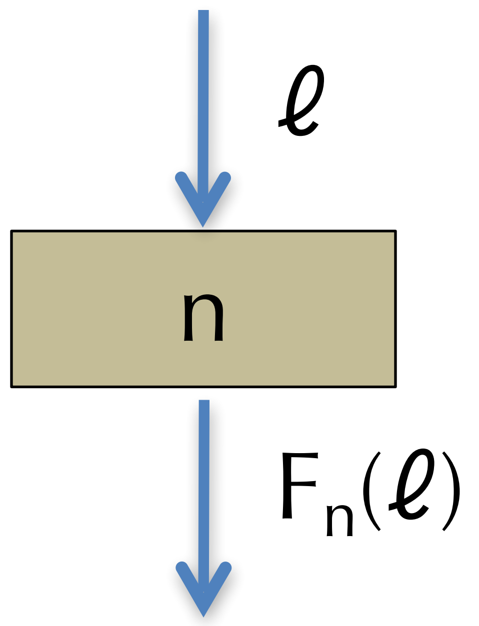
Common Features

- Liveness and Possible Values Analyses had similarities
  - In both, we have some **domain** of information we attach to each point in the program
    - Liveness, the domain is sets of variables
    - Possible values, the domain is maps from variables to (abstractions of sets of values)
- Each analysis has a notion of flow function
  - How do we update the information based on each operation in the program.
- But they propagate information in opposite directions
  - Liveness is Backwards: if I use a variable now, it is live in previous program points
  - Possible values is Forwards: if I learn a variables value now, I know it later as well
- Each analysis aggregates information at control flow boundaries
  - Liveness takes the union of successors at a conditional branch
  - Possible values takes the union of predecessors at a block/function
- Perform the analysis by starting from incorrect information and iterating until we get the same result, a fixed point.

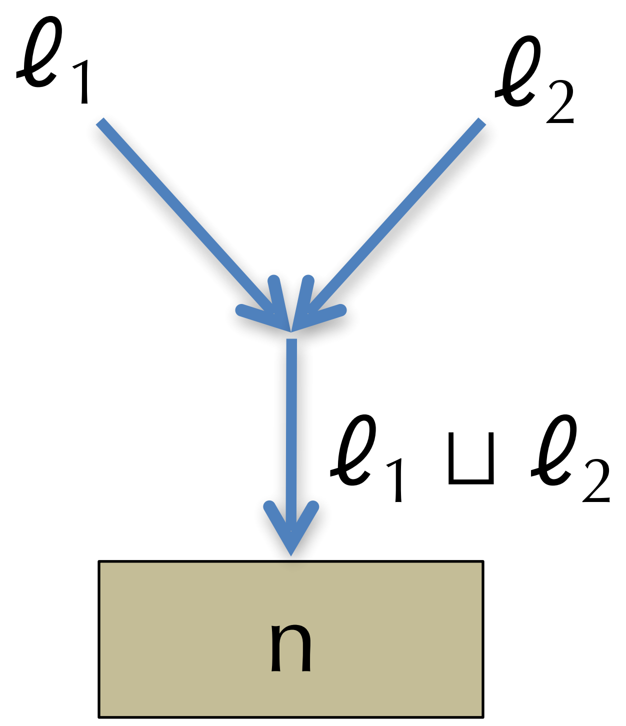
(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values L
   - e.g. L = the powerset of all variables
   - Think of $\ell \in L$ as a property, then "$x \in \ell$" means "x has the property"



2. For each node n, a flow function $F_n : L \to L$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A combining operator $\sqcup$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \sqcup \ell_2$"
   - $in[n] := \bigsqcup_{n' \in pred[n]} out[n']$



Generic Iterative (Forward) Analysis

for all n, $in[n] := \bot$, $out[n] := \bot$

repeat until no change

  for all n

    $in[n] := \bigsqcup_{n' \in pred[n]} out[n']$

    $out[n] := F_n(in[n])$

  end

end

- Here, $\bot \in L$ ("bottom") represents having the "most precise" constraint
  - Having "more precise" information enables more optimizations
  - "most precise" amount could be inconsistent with the constraints.
  - Iteration refines the answer, eliminating inconsistencies, producing less precise results

Structure of L

- The domain has structure that reflects the "amount" of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever ℓ_2 provides at least as much information as ℓ_1.
  - The dataflow value ℓ_2 is "better" for enabling optimizations.

- Example 1: for liveness and possible values analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \supseteq \ell_2$

- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$

$\mathcal{L}$ as a Partial Order

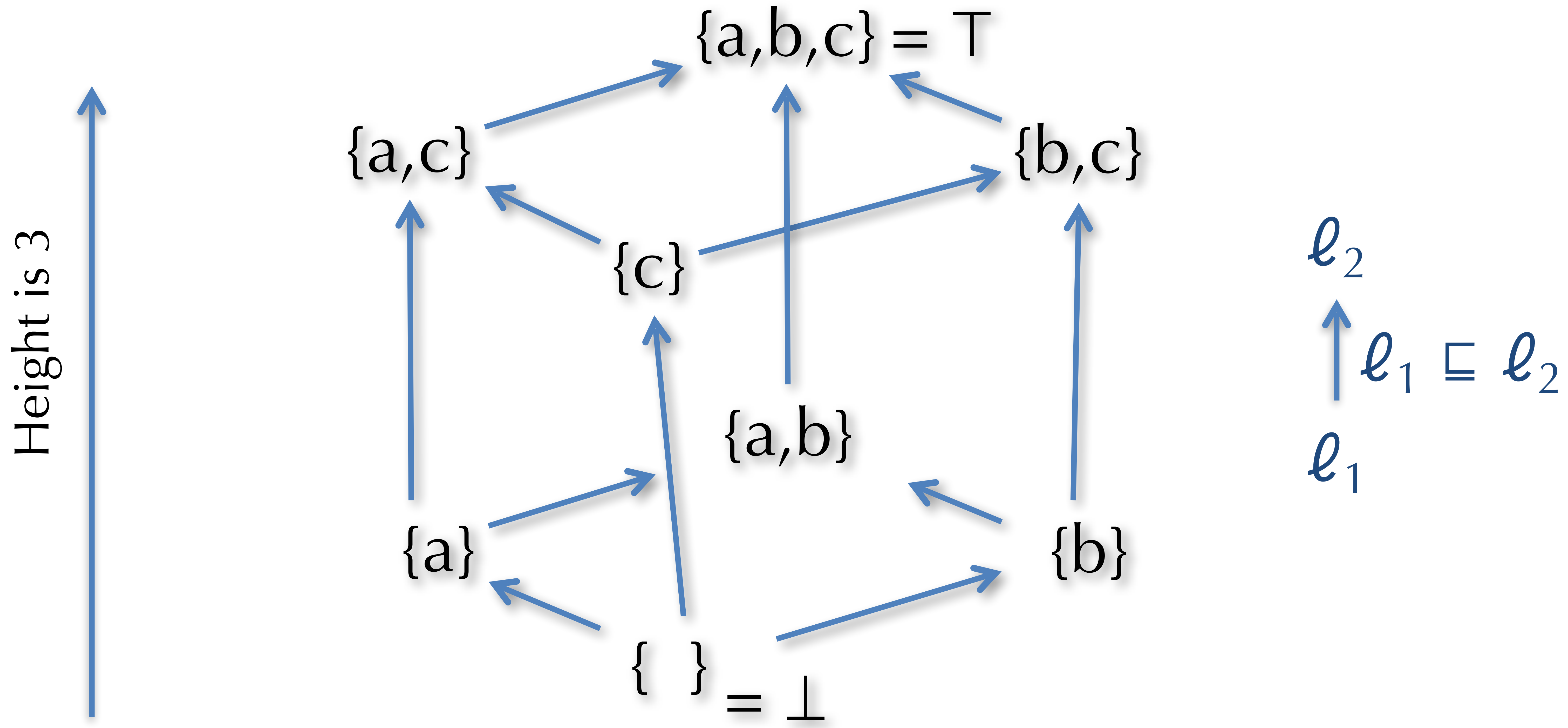
- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq \ell_1$

- Properties of a partial order:
  - *Reflexivity:* $\ell \sqsubseteq \ell$
  - *Transitivity:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ implies $\ell_1 \sqsubseteq \ell_2$
  - *Anti-symmetry:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$ implies $\ell_1 = \ell_2$

- Examples:
  - Integers ordered by $\leq$
  - Sets ordered by $\subseteq$ or $\supseteq$

Subsets of {a,b,c} ordered by ⊆

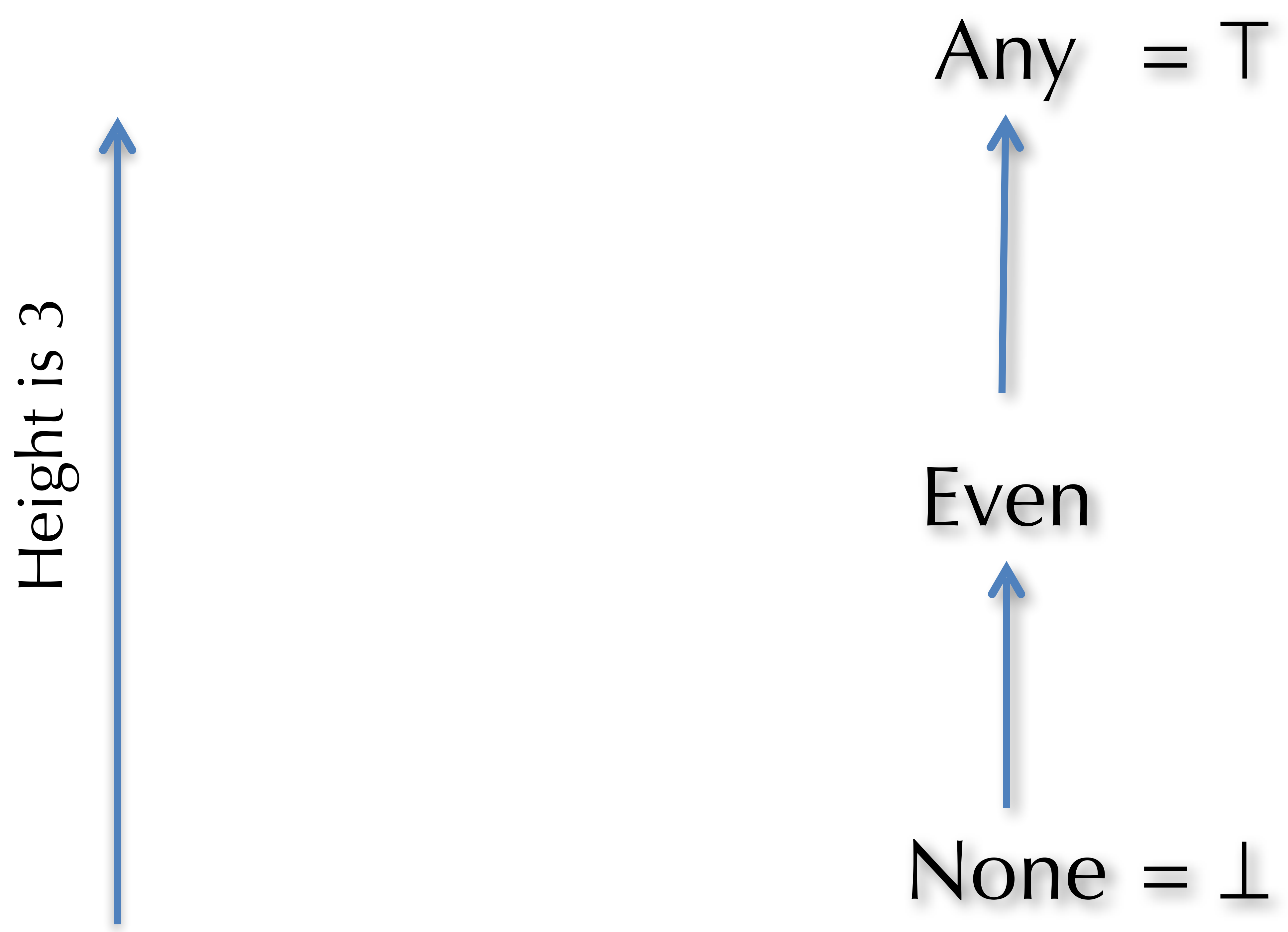
Partial order presented as a Hasse diagram.



order $\sqsubseteq$ is $\subseteq$ meet $\sqcap$ is $\cap$ join $\sqcup$ is $\cup$

Possible Values

Partial order presented as a Hasse diagram.



$$\text{Any} \sqcup \text{Any} = \text{Any} \sqcup \text{Even} = \text{Any} \sqcup \text{None} = \text{Any}$$

$$\text{Even} \sqcup \text{Even} = \text{Even} \sqcup \text{None} = \text{Even}$$

$$\text{None} \sqcup \text{None} = \text{None}$$

Meets and Joins

- The combinig operator, $\sqcup$ operator is called the "join" operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$
    "the join is an upper bound"
  - If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$
    "there is no smaller upper bound"

- The dual operator $\sqcap$ is called the "meet" operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$
    "the meet is a lower bound"
  - If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \sqcap \ell_2$
    "there is no greater lower bound"

- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it's called a *meet semi-lattice*.

Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
- $out[n] := F_n(in[n])$
- Equivalently: $out[n] := F_n(\bigsqcup_{n' \in pred[n]} out[n'])$
  - By definition of in[n]
- We can write this as a simultaneous update of the vector of out[n] values:
  - let $x_n = out[n]$
  - Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcup_{j \in pred[1]} out[j]), F_2(\bigsqcup_{j \in pred[2]} out[j]), \ldots, F_n(\bigsqcup_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

Iteration Computes Fixpoints

- Let $\mathbf{X}_0 = (\perp, \perp, \ldots, \perp)$
- Each loop through the algorithm apply F to the old vector:
  $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$
  $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$
  …
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^k(\mathbf{X}))$
- A fixpoint is reached when $\mathbf{F}^k(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$
  - That's when the algorithm stops.

- Wanted: a minimal fixpoint
  - Because that one is more informative/useful for performing optimizations

Monotonicity & Termination

- Each flow function F_n maps lattice elements to lattice elements; to be sensible is should be *monotonic*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$ is *monotonic* iff:
  $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
  - Intuitively: "If you have more information entering a node, then you have more information leaving the node."
- Monotonicity lifts point-wise to the function: $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  - vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each i
- Since we start at, each iteration moves up the lattice that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\perp \sqsubseteq \mathbf{F}(\perp) \sqsubseteq \mathbf{F}(\mathbf{F}(\perp)) \sqsubseteq \ldots$
- Therefore, # steps needed to reach a fixpoint is at most the height H of $\mathcal{L}$ times the number of nodes: O(Hn)

“Classic” Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

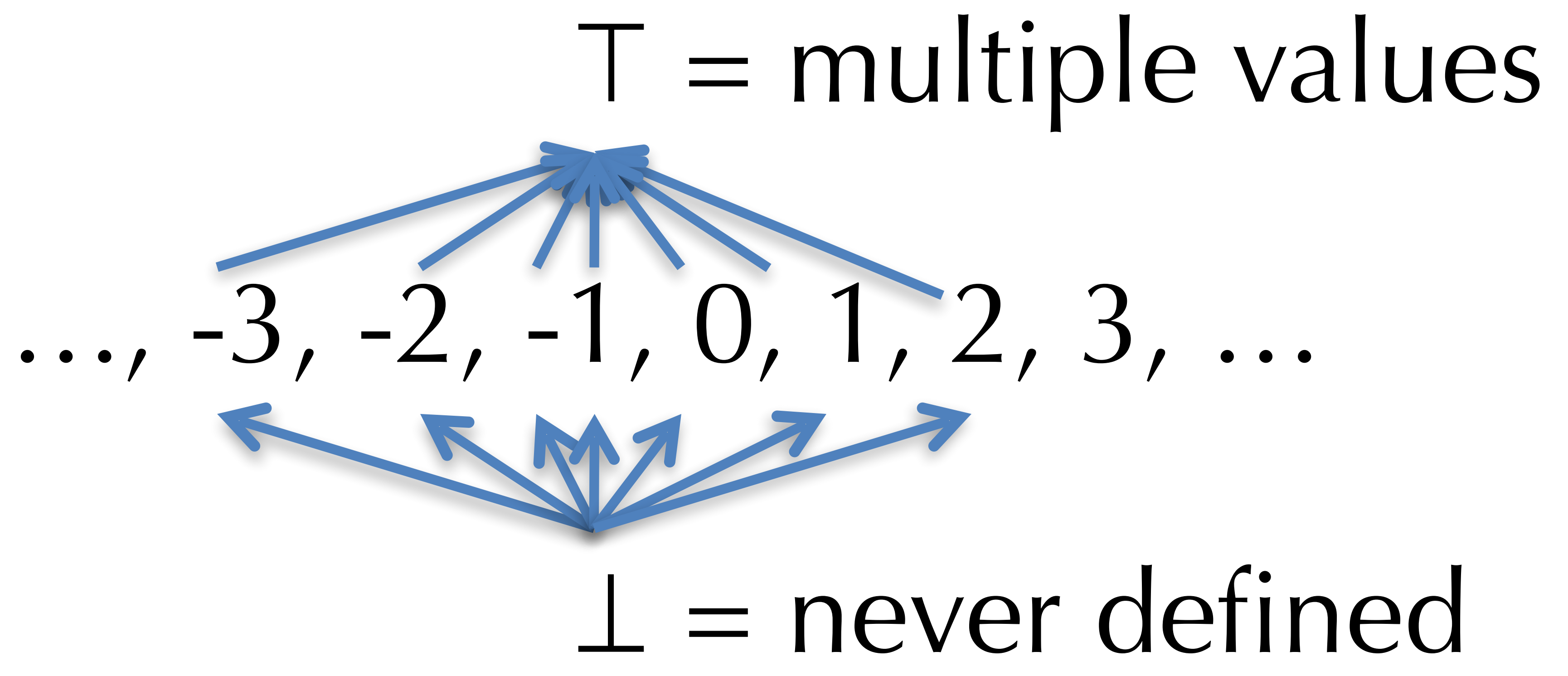
- Idea: propagate and fold integer constants in one pass:

x = 1;		x = 1;
y = 5 + x;	⇒	y = 6;
z = y * y;		z = 36;

- Information about a single variable:
  - Variable is never defined.
  - Variable has a single, constant value.
  - Variable is assigned multiple values.

Domains for Constant Propagation

- We can make a constant propagation lattice L for *one variable* like this:



- To accommodate multiple variables, we take the product lattice, with one element per variable.
  - Assuming there are three variables, x, y, and z, the elements of the product lattice are of the form (ℓ_x, ℓ_y, ℓ_z).
  - Alternatively, think of the product domain as a context that maps variable names to their "*abstract interpretations*"
- What are "meet" and "join" in this product lattice?
- What is the height of the product lattice?

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.