CS 477: Dataflow Analysis and Abstract Interpretation

Sasa Misailovic

Based on previous slides by Martin Vechev

University of Illinois at Urbana-Champaign

General Sources of Imprecision

Abstraction Imprecision

- Concrete values (integers) abstracted as lattice values (e.g., -,0, and +)
- Lattice values less precise than execution values
- Abstraction function throws away information

Control Flow Imprecision

- One lattice value for all possible control flow paths
- Analysis result has a single lattice value to summarize results of multiple concrete executions
- Join operation $\vee$ moves up in lattice to combine values from different execution paths
- Typically if $x \leq y$, then x is more precise than y

Why To Allow Imprecision?

Make analysis tractable

Unbounded sets of values in execution

- Typically abstracted by finite set of lattice values

Execution may visit unbounded set of states

- Abstracted by computing joins of different paths

Intuition Behind Abstract Interpretation

Patrick Cousot's Description:

- http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Also, Abstract interpretation vs Dataflow Analysis:

- Comparison with CS 701-style Dataflow Analysis: http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/10.ABSTRACT-INTERPRETATION.html#701
- Systematic design of program analysis frameworks: https://www.di.ens.fr/~cousot/COUSOTpapers/POPL79.shtml
- Program Analysis as Model Checking of Abstract Interpretations: http://web.cs.ucla.edu/~palsberg/course/purdue/cs661/F01/papers/schmidt-steffen-sas98.pdf
- Introduction to abstract interpretation: http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/PAPERS/abstractInterp.Rosendahl.pdf

Summary of formal program analysis:

- http://www.kroening.com/papers/tcad-sw-2008.pdf

Comparison with CS 701-style Dataflow Analysis

How does abstract interpretation compare with the kind of dataflow analysis studied in CS 701? One thing that may look like a significant difference but in fact is not, is the way the analyses are actually carried out. In 701, we did the following:

- Define a complete lattice L with no infinite descending chains. The elements of L are the dataflow facts.
- Specify one lattice element as the special "initial" value.
- For each CFG edge n→m, define a monotonic function $f_{n \to m}$ of type L → L. The function for the edge out of the enter node ignores its input and produces the special initial value as its output. (Note: We sometimes defined the functions on CFG nodes rather than on CFG edges. There are examples where putting the functions on the nodes is more convenient, and other examples where putting the functions on the edges is more convenient. In both cases, the functions capture the same semantics, so there is not really a significant difference.)
- Define a cross-product lattice LX whose tuples have as many items as there are CFG nodes. Also define a (monotonic) function F of type LX → LX, that uses functions $f_{m \to n}$ to define the n^{th} "slot" of a tuple.
- To solve a dataflow problem, ***iterate down from top***; i.e., start with the top element of lattice LX, then apply function F repeatedly until there is no change.

Abstract interpretation is similar: To ensure termination of an iterative algorithm, the abstract domain must be a complete lattice with no infinite *ascending* chains, and the abstract transfer functions must still be monotonic. The solution is computed by iterating up from bottom, instead of down from top, but since a complete lattice is "symmetric", this is just a cosmetic difference.

The real difference between the two approaches is that for 701-style dataflow analysis, we define the lattice elements and the CFG-node dataflow functions based only on intuition. Therefore, there is no guarantee that the solution to a dataflow problem has any relationship to the program's semantics. In contrast, since part of abstract interpretation involves showing relationships between the concrete and abstract semantics, we do have such guarantees.

The price we pay is that it is not always clear how to define dataflow problems of interest using abstract interpretation. For example, problems like reaching definitions require knowing more than just the sets of states that can arise at each CFG node: we also need to know which other CFG nodes assigned the values to the variables. This is usually done by defining an *instrumented* collecting semantics, which keeps additional information (like the label of the CFG node that most recently assigned to a variable) in a state. While this allows reaching definitions to be defined, it may seem rather ad hoc.

A similar issue arises with backward problems like live variable analysis. One interesting approach to defining the live-variables problem using abstract interpretation and continuation semantics is given in a set of lecture notes called Introduction to Abstract Interpretation by Mads Rosendahl.

Abstract Interpretation

1. Define abstract domains (that represent important parts of program execution)

2. Define abstraction and concretization functions to relate the abstract domain with the program execution

3. Iterate updating the abstract state until convergence

The Art of Sound* Approximation: Static Program Analysis

- Define a function $F^\sharp$ such that $F^\sharp$ approximates F. This is typically done manually and can be tricky but is done once for a particular programming language.

- Then, use existing theorems which state that the least fixed point of $F^\sharp$, e.g. denote it V, approximates the least fixed point of F, e.g. denote it $[\![P]\!]$

- Finally, automatically compute a fixed point of $F^\sharp$, that is a V where $F^\sharp(V) = V$

* For a reminder and discussion about soundness and precision, see these articles:
http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/
https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext

Abstract Interpretation: step-by-step

1. Select/define an abstract domain
   - selected based on the type of **properties** you want to prove

2. Define abstract semantics **for the language** w.r.t. to the domain
   - prove **sound** w.r.t **concrete semantics**
   - involves defining abstract transformers
     - that is, effect of statement / expression on the abstract domain

3. Iterate abstract transformers over the abstract domain
   - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

FUNCTION APPROXIMATION

Approximating a Function

Given functions:

$$F: C \rightarrow C$$

$$F^{\sharp}: C \rightarrow C$$

what does it mean for $F^{\sharp}$ to approximate F ?

$$\forall x \in C : F(x) \sqsubseteq_c F^{\sharp}(x)$$

Approximating a Function

What about when:

$$F: C \rightarrow C$$

$$F^{\sharp}: A \rightarrow A$$

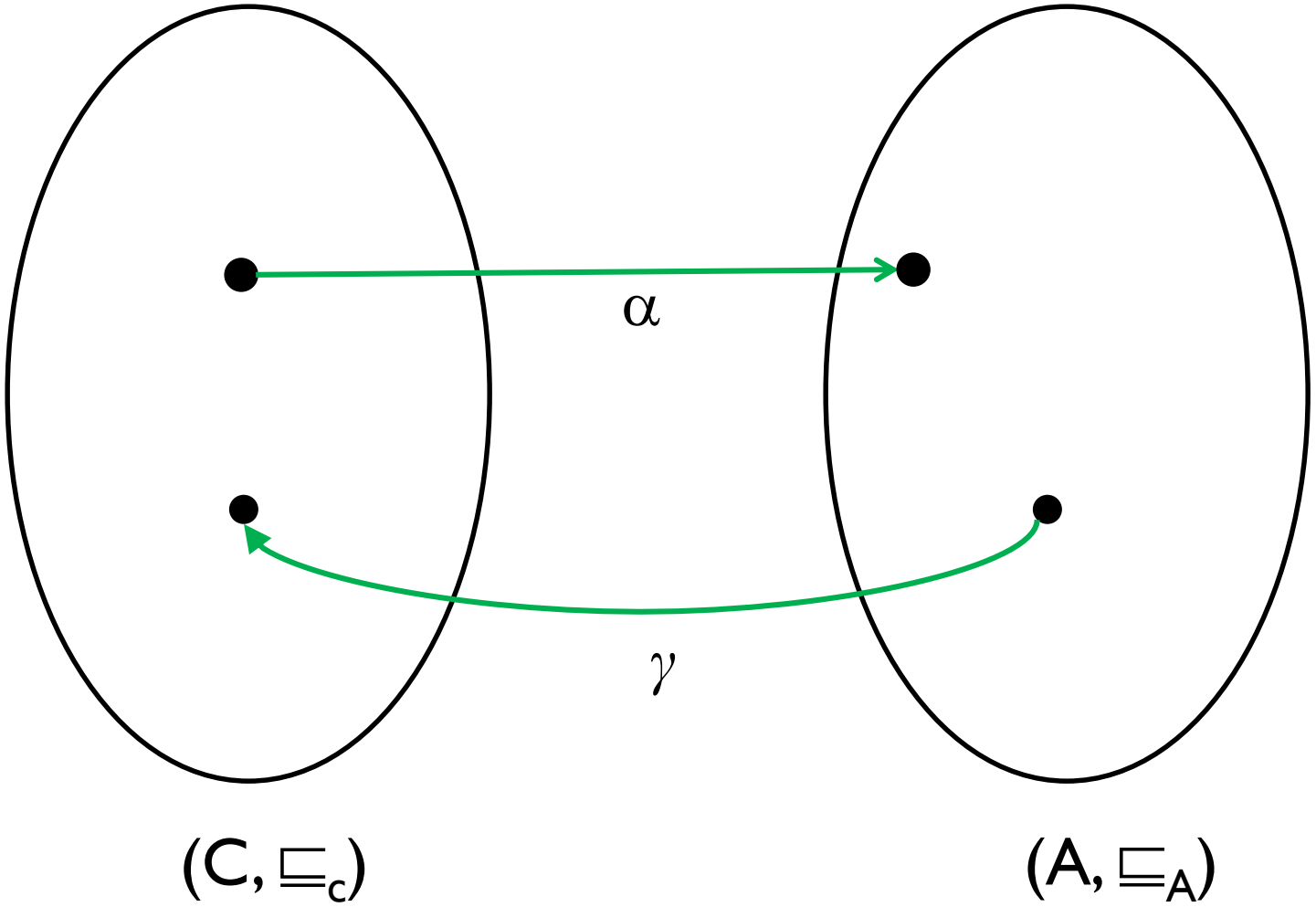
We need to connect the concrete C and the abstract A

We will connect them via two functions α and γ

$\alpha : C \rightarrow A$ is called the **abstraction** function

$\gamma : A \rightarrow C$ is called the **concretization** function

Connecting Concrete with Abstract



Approximating Function: Definition I

So we have the 2 functions:

$$F: C \rightarrow C$$

$$F^{\sharp}: A \rightarrow A$$

If we know that α and γ form a **Galois Connection**, then we can use the following definition of approximation:

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^{\sharp}(z)$$

Galois Connection

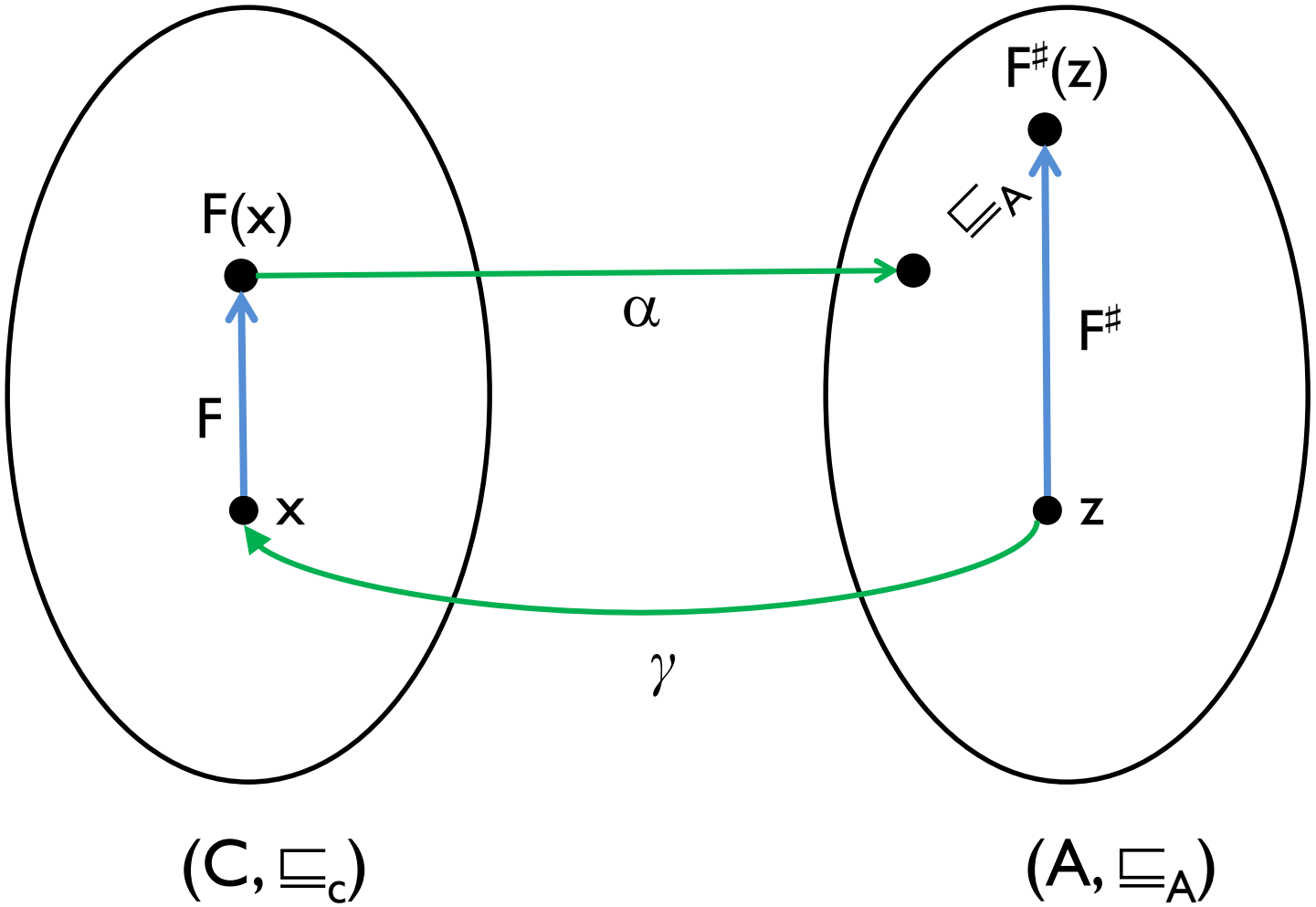
For the course, **it is not important** to know precisely what Galois Connections are.

The only point to keep in mind that is that they place some **restrictions** on α and γ.

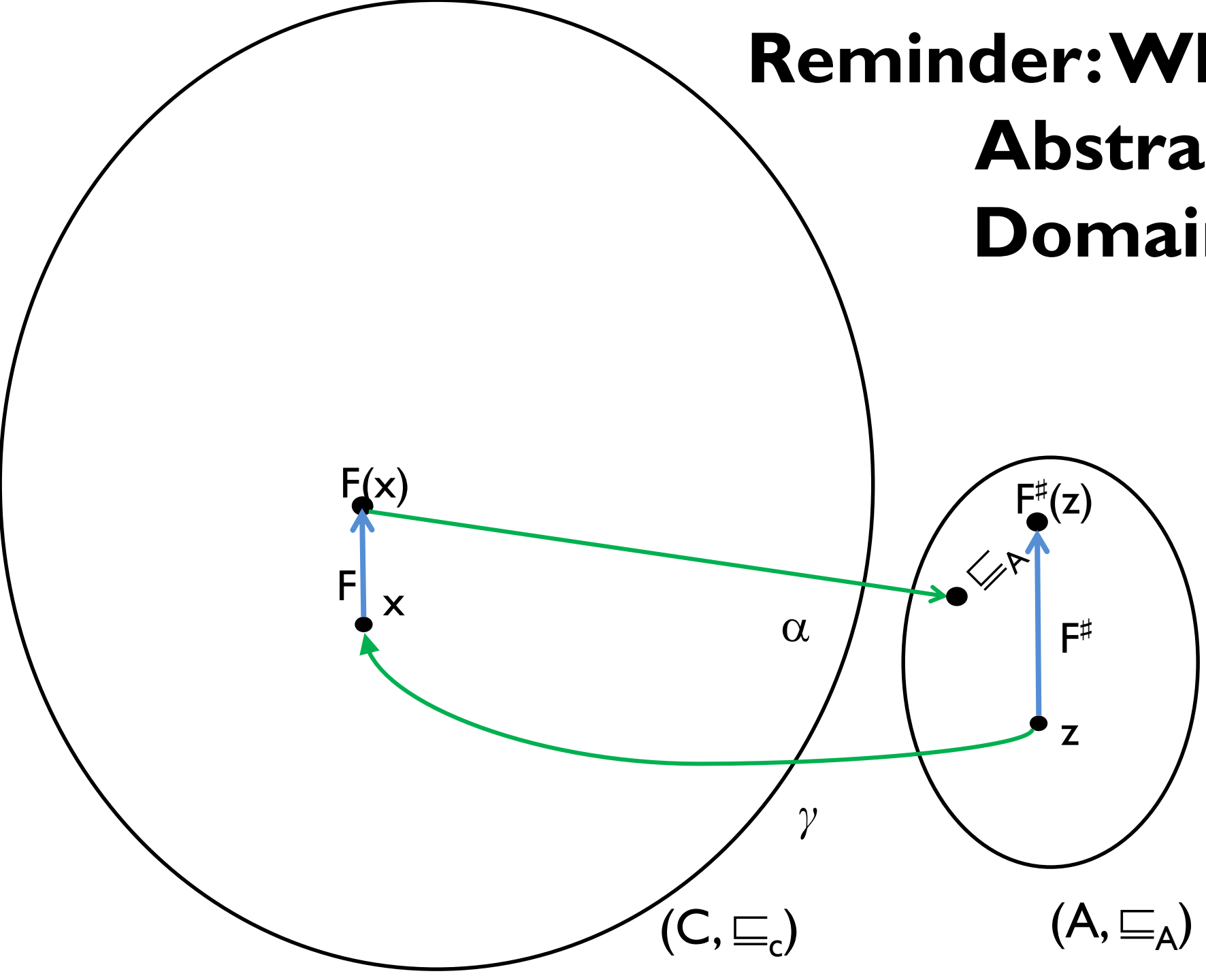
- For instance, they **require α to be monotone**.

Visualizing Definition 1

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\sharp(z)$$



Reminder: Why Abstract Domain?



Approximating a Function

This equation

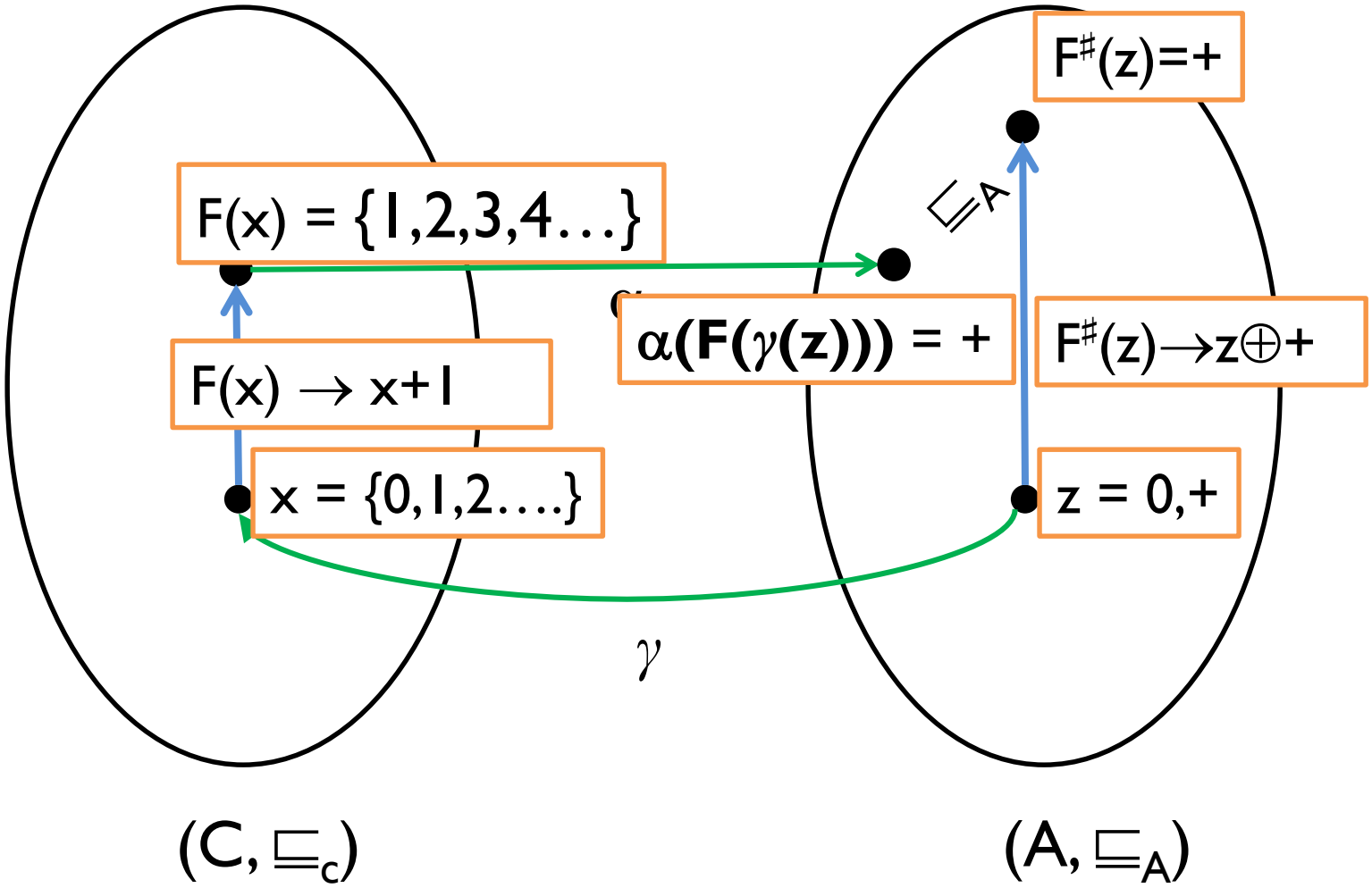
$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\sharp(z)$$

says that

- if we have some function in the abstract domain that we think **should approximate** the concrete function,
- then to check that this is indeed true, we need to prove
- that for any abstract element, (1) concretizing it, (2) applying the concrete function and (3) abstracting back again is **less than** applying the function in the abstract directly.

Visualizing Definition 1

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^{\sharp}(z)$$



Least precise approximation

To approximate F, we can always define $F^{\sharp}(z) = T$

This solution is always **sound** as: $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A T$

However, it is not practically useful as it is **too imprecise**

Most precise approximation

$F^{\sharp}(z) = \alpha(F(\gamma(z)))$ is the **best abstract function.**

But, we often **cannot implement** best $F^{\sharp}(z)$ algorithmically.

However, we can come up with a $F^{\sharp}(z)$ that has the **same behavior** as $\alpha(F(\gamma(z)))$ but a **different implementation**.

Any such $F^{\sharp}(z)$ is referred to as the **best transformer**.

Key Theorem I: Least Fixed Point Approximation

If we have:

1. **monotone** functions $F: C \rightarrow C$ and $F^{\sharp}: A \rightarrow A$
2. $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forming a Galois Connection
3. $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^{\sharp}(z)$ (that is, $F^{\sharp}$ approximates F)

then:

$$\alpha\ (\mathrm{lfp}(F)) \sqsubseteq_A \mathrm{lfp}\ (F^{\sharp})$$

This is important as it goes from **local** function approximation to **global** approximation. This is a key theorem in program analysis.

Least Fixed Point Approximation

The 3 premises to the theorem are usually proved **manually**.

Once proved, we can now **automatically** compute a least fixed point in the abstract and be sure that our result is **sound** !

So what is $F^\sharp$ then ?

$F^\sharp$ is to be defined for the particular abstract domain **A** that we work with. The domain **A** can be Sign, Parity, Interval, Octagon, Polyhedra, and so on.

In our setting and commonly, we simply keep a map from every label (program counter) in the program to an abstract element in **A**

Then $F^\sharp$ simply updates the mapping from labels to abstract elements.

$F^\sharp$

$$F^\sharp : (\mathtt{Lab} \rightarrow \mathtt{A}) \rightarrow (\mathtt{Lab} \rightarrow \mathtt{A})$$

$$F^\sharp(m)\ell = \begin{cases} \top & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell',\text{action},\ell)} [\![\text{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$$

$[\![\text{action}]\!] : \mathtt{A} \rightarrow \mathtt{A}$

$[\![\text{action}]\!]$ is the key ingredient here. It captures the effect of a language statement on the abstract domain $\mathtt{A}$. Once we define it, we have $F^\sharp$

$[\![\text{action}]\!]$ is often referred to as the **abstract transformer**.

What is (ℓ',action, ℓ) ?

```
foo (int i) {

1: int x := 5;
2: int y := 7;

3: if (0 ≤ i) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }
7:}
```

Actions:

```
(1, x := 5, 2)
(2, y := 7, 3)
(3, 0 ≤ i, 4)
(3, 0 > i, 7)
(4, y = y + 1, 5)
(5, i := i – 1, 6)
(6, goto 3, 3)
```

Multiple (two) actions reach label 3

What is action ?

An **action** can be:

- b ∈ BExp
- x:= a
- skip

boolean expression in a conditional
here, a ∈ AExp

In performing an action, the assignment and the boolean expression of a conditional is **fully evaluated**

```
                x:=y+x                                  if (x > 5) …
{x↦2, y↦0}      →      {x↦4, y↦0}      {x↦2, y↦0}      →
```

Defining ⟦action⟧

⟦action⟧ captures the abstract semantics of the language for a particular abstract domain.

We will see precise definitions for some actions in the Interval domain. Defining ⟦action⟧ for complex domains such as Octagon can be quite tricky.

Cheat Sheet: Connecting Math and Analysis

Mathematical Concept	Use in Static Analysis
Complete Lattice	Defines Abstract Domain and ensure joins exist.
Joins ($\sqcup$)	Combines facts arriving at a program point
Bottom ($\bot$)	Used for initialization of all but initial elements
Top (T)	Used for initialization of initial elements
Widening ($\triangledown$)	Used to guarantee analysis termination
Function Approximation	Critical to make sure abstract semantics approximate the concrete semantics
Fixed Points	This is what is computed by the analysis
Tarski's Theorem	Ensures fixed points exist.

Checkpoint

So far, we have seen a bunch of mathematical concepts such as lattices, functions, fixed points, function approximation, etc.

Next, we will see how to put these together in order to build static analyzers.

Domain of Program States

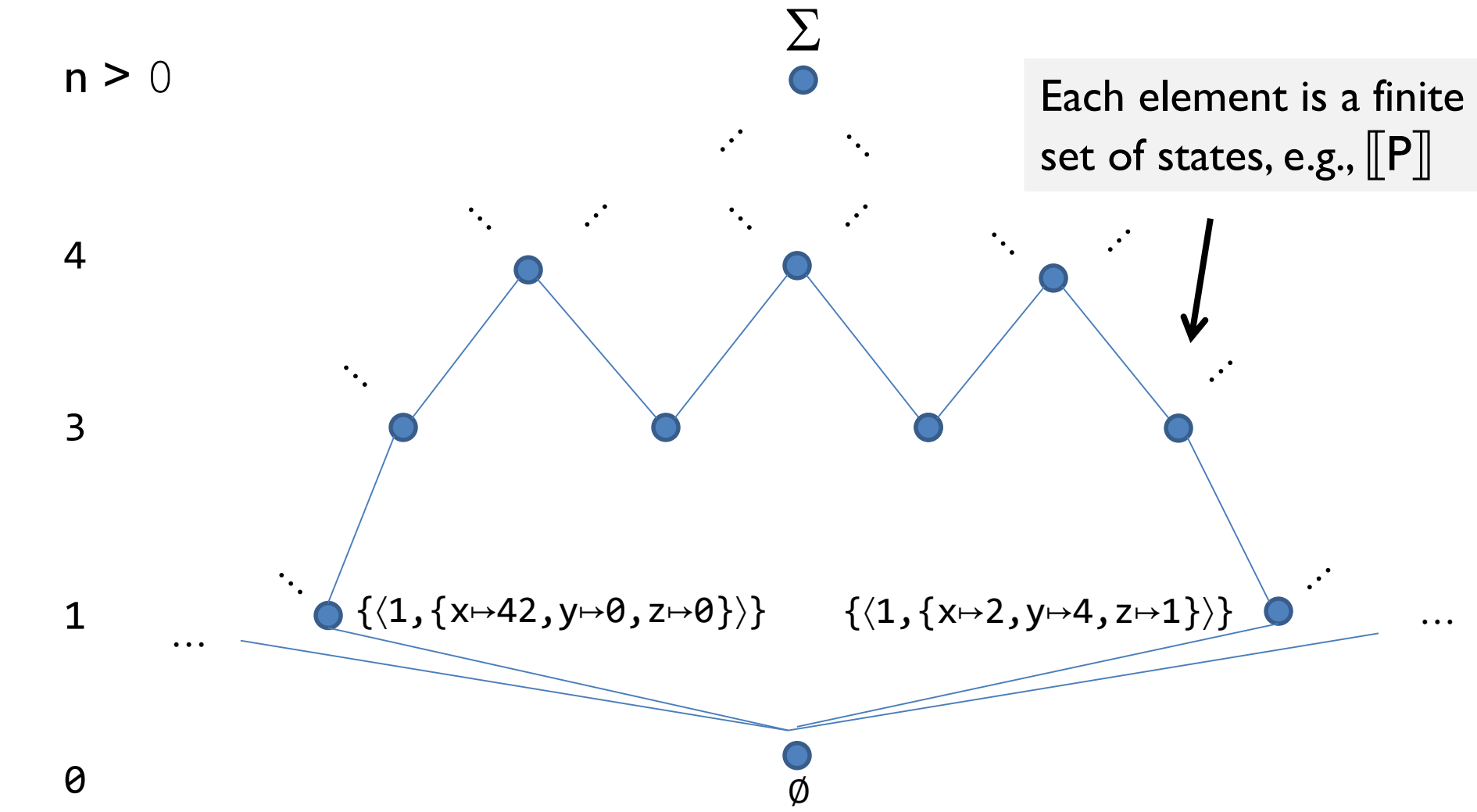
Our starting point is a domain where each element of the domain is a **set of states**. The domain of states is a **complete lattice**:

$$(\wp(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma)$$

$$\Sigma = \text{Label} \times \text{Store}$$

Starting Point: Domain of States

Size of Set:



Representing $[\![P]\!]$

Let $[\![P]\!]$ be the **set of reachable states** of a program P.

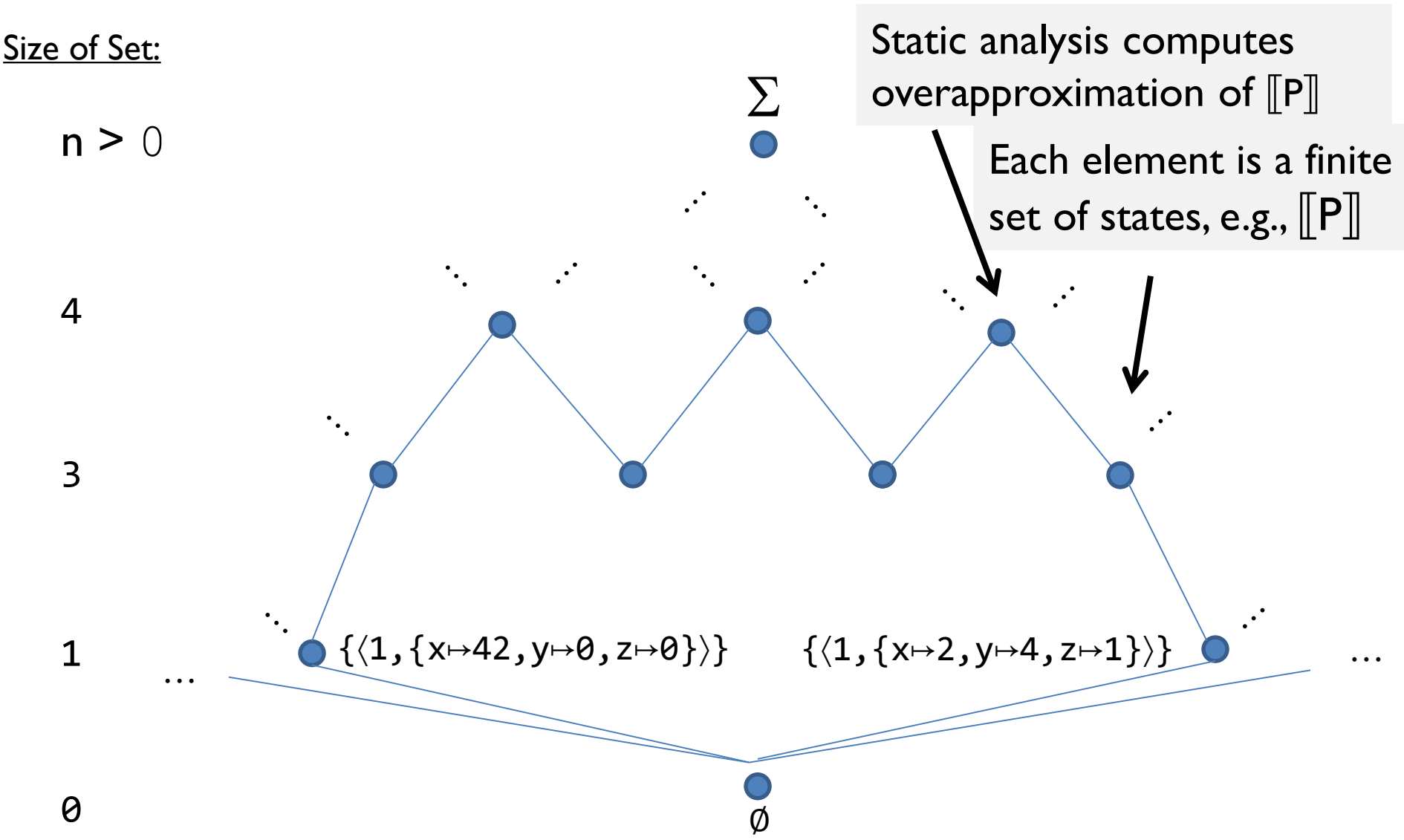
Def. Let function F be (where I is an initial set of states):

$$F(S) = I \cup \{ c' \mid c \in S \wedge c \rightarrow c' \}$$

Then, $[\![P]\!]$ is a **fixed point** of F: i.e., $F([\![P]\!]) = [\![P]\!]$

(in fact, $[\![P]\!]$ is the least fixed point of F)

Starting Point: Domain of States



Abstract Interpretation: step-by-step

1. select/define an abstract domain
   - selected based on the type of **properties** you want to prove

2. define abstract semantics **for the language** w.r.t. to the domain
   - prove **sound** w.r.t **concrete semantics**
   - involves defining abstract transformers
     - that is, effect of statement / expression on the abstract domain

3. iterate abstract transformers over the abstract domain
   - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

Abstract Interpretation: Step 1

1. select/define an abstract domain
   - selected based on the type of **properties** you want to prove

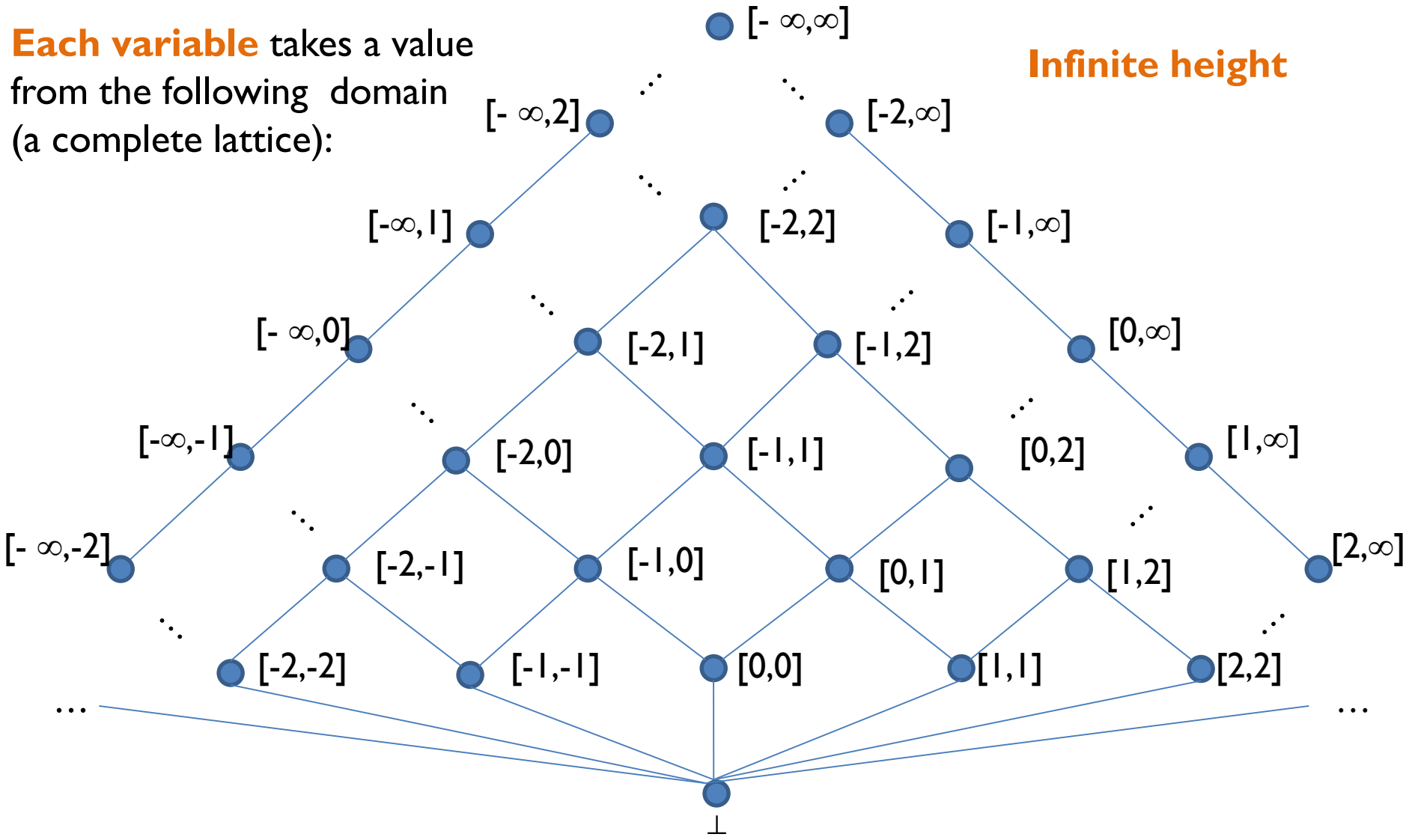
Interval Domain

If we are interested in properties that involve the range of values that a variable can take, we can abstract the set of states into a map which captures the range of values that a variable can take.

Interval Domain

Each variable takes a value from the following domain (a complete lattice):

Infinite height



Interval Domain: Lets Define it

Let the interval domain **on integers** be a lattice: $(L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, [-\infty,\infty])$

We denote $\mathbf{Z^\infty} = Z \cup \{-\infty,\infty\}$
The set $\mathbf{L^i} = \{[x,y] \mid x,y \in Z^\infty,\ x \leq y\} \cup \{\bot_i\}$

For a set $S \subseteq Z^\infty$, `min(S)` returns the minimum number in `S`, `max(S)` returns the maximum number in `S`.

Operations ($\sqsubseteq_i$, $\sqcup_i$, $\sqcap_i$):

- $[a,b] \sqsubseteq_i [c,d]$ if $c \leq a$ and $b \leq d$
- $[a,b] \sqcup_i [c,d] = [\min(a,c), \max(b,d)]$
- $[a,b] \sqcap_i [c,d] =$ `meet(max(a,c), min(b,d))`
  where `meet(x,y)` returns `[x,y]` if $x \leq y$ and $\bot_i$ otherwise

Intervals: Applied to Programs

The $\mathtt{L}^{\mathtt{i}}$ domain simply defines intervals, but to apply it to programs we need to take into account program labels (program counters) and program variables.

Therefore, for programs, **we use the domain** $\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^{\mathtt{i}})$

That is, at each label and for each variable, we will keep the range for that variable. This domain is also a **complete lattice.**

The operators of $\mathtt{L}^{\mathtt{i}}$ $\sqsubseteq_{\mathtt{i}}, \sqcup_{\mathtt{i}}, \sqcap_{\mathtt{i}}$ are **lifted directly** to both domains:

- $\mathtt{Var} \rightarrow \mathtt{L}^{\mathtt{i}}$
- $\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^{\mathtt{i}})$

Intervals: Applied to Programs

$$\alpha^{\mathtt{i}}: \quad \wp(\Sigma) \to (\mathtt{Lab} \to (\mathtt{Var} \to \mathtt{L}^{\mathtt{i}}))$$
$$\gamma^{\mathtt{i}}: \quad (\mathtt{Lab} \to (\mathtt{Var} \to \mathtt{L}^{\mathtt{i}})) \to \wp(\Sigma)$$

Using $\alpha^{\mathtt{i}}$, we abstract a set of states into a map from program labels to interval ranges for each variable.

Using $\gamma^{\mathtt{i}}$, we concretize the intervals maps to a set of states

Example of Abstraction and Concretization

α^i ({ ⟨1,{x↦1,y↦9,q↦-2}⟩, ⟨1,{x↦5,y↦9,q↦-2}⟩, ⟨1,{x↦8,y↦9,q↦-2}⟩,
⟨1,{x↦1,y↦9,q↦4}⟩, ⟨1,{x↦5,y↦9,q↦4}⟩, ⟨1,{x↦8,y↦9,q↦4}⟩}
)
= 1 → (x ↦ [1,8], y ↦ [9,9], q ↦ [-2,4])

γ^i (1 → (x ↦ [1,8], y ↦ [9,9], q ↦ [-2,4])

= { ⟨1,{x↦1,y↦9,q↦-2}⟩, ⟨1,{x↦5,y↦9,q↦-2}⟩, ⟨1,{x↦8,y↦9,q↦-2}⟩,
⟨1,{x↦1,y↦9,q↦4}⟩, ⟨1,{x↦5,y↦9,q↦4}⟩, ⟨1,{x↦8,y↦9,q↦4}⟩,
⟨1,{x↦7,y↦9,q↦3}⟩, ⟨1,{x↦3,y↦9,q↦4}⟩, ⟨1,{x↦1,y↦9,q↦-1}⟩,
…, …, … }

Concretization includes many more states (in red) than what was abstracted…

Abstract Interpretation: Step 2

1. select/define an abstract domain
   - selected based on the type of **properties** you want to prove

2. define abstract semantics **for the language** w.r.t. to the domain
   - prove **sound** w.r.t **concrete semantics**
   - involves defining abstract transformers
     - that is, effect of statement / expression on the abstract domain

we still need to actually compute α^i ($[\![P]\!]$)
(or an over-approximation of it)

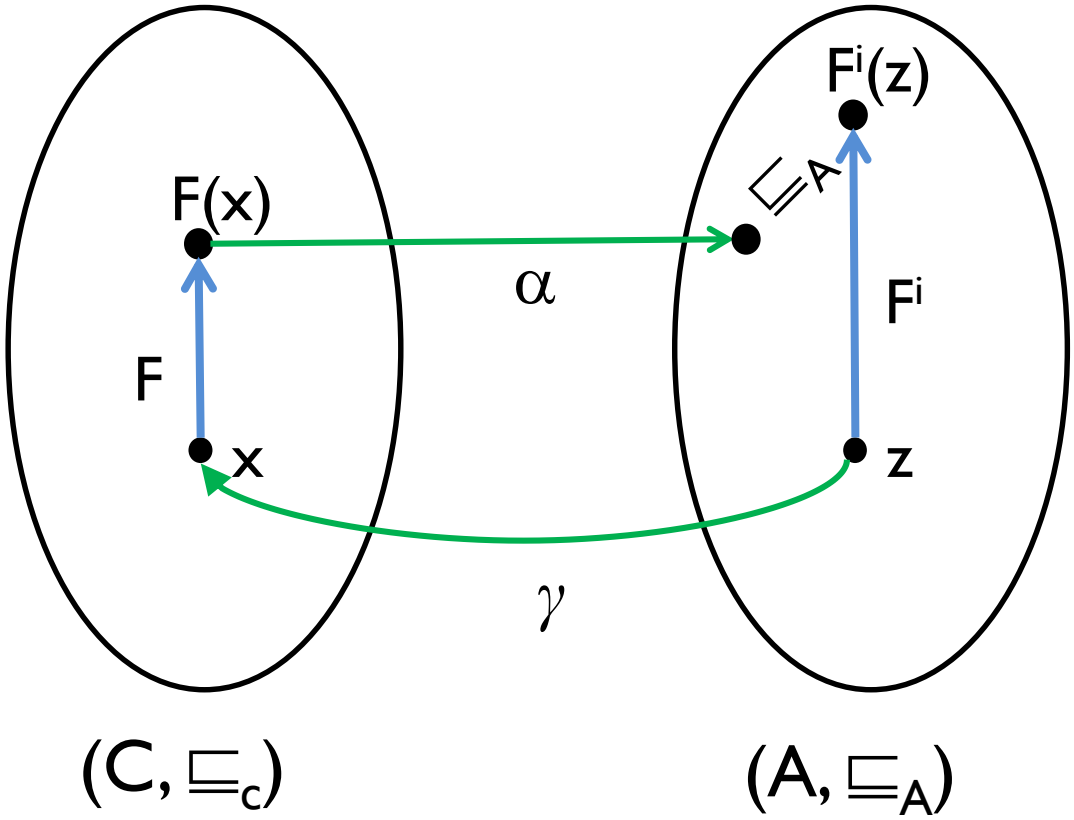
We need to approximate F

We want a function F^i where:

$$F^i : (\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^i)) \rightarrow (\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^i))$$

such that:

$\alpha^i\ (\mathbf{lfp}\ F) \sqsubseteq\ \mathbf{lfp}\ F^i$



Lets define F^i

$F^i : (\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^i)) \rightarrow (\mathtt{Lab} \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^i))$

Here is a definition of F^i which approximates the best transformer but works only on the abstract domain:

$$F^i(m)\ell = \begin{cases} \lambda\, v.\, [-\infty, \infty] & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', action, \ell)} [\![action]\!]_i(m(\ell')) & \text{otherwise} \end{cases}$$

$[\![action]\!]_i : (\mathtt{Var} \rightarrow \mathtt{L}^i) \rightarrow (\mathtt{Var} \rightarrow \mathtt{L}^i)$

What is (ℓ',action, ℓ) ?

```
foo (int i) {

1: int x := 5;
2: int y := 7;

3: if (0 ≤ i) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }
7:}
```

Actions:

```
(1, x := 5, 2)
(2, y := 7, 3)
(3, 0 ≤ i, 4)
(3, 0 > i, 7)
(4, y = y + 1, 5)
(5, i := i – 1, 6)
(6, goto 3, 3)
```

Multiple (two) actions reach label 3

What is $(\ell', \text{action}, \ell)$?

- $(\ell', \text{action}, \ell)$ is an edge in the control-flow graph

- More formally, if there exists a transition $t = \langle \ell', \sigma' \rangle \rightarrow \langle \ell, \sigma \rangle$ in a program trace in P, where t was performed by statement called action, then $(\ell', \text{action}, \ell)$ must exist. This says that we are sound: we never miss a flow.

- However, $(\ell', \text{action}, \ell)$ may exist even if no such transition t above occurs. In this case, the analysis would be imprecise as we would unnecessarily create more flows.

What is $(\ell', \text{action}, \ell)$?

An action can be:

- $b \in \mathtt{BExp}$ — boolean expression in a conditional
- $x := a$ — here, $a \in \mathtt{AExp}$
- skip

Next, we will define the effect of some of these things formally, while with others we will proceed by example.

The key point is to make sure that $[\![\text{action}]\!]_i$ produces sound and precise results.

F^i on an example

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

$F^i(m)1 = \quad \lambda\ v.\ [-\infty,\infty]$

$F^i(m)2 = \quad [\![x := 5]\!]_i\,(m(1))$

$F^i(m)3 = \quad [\![y := 7]\!]_i\,(m(2)) \sqcup [\![goto\ 3]\!]_i\,(m(6))$

$F^i(m)4 = \quad [\![i \geq 0]\!]_i\,(m(3))$

$F^i(m)5 = \quad [\![y := y + 1]\!]_i\,(m(4))$

$F^i(m)6 = \quad [\![i := i - 1]\!]_i\,(m(5))$

$F^i(m)7 = \quad [\![i < 0]\!]_i\,(m(3))$

$[\![x := a]\!]_i$

$$[\![x := a]\!]_i\,(m) = m\,[x \mapsto v] \quad , \quad \text{where } \langle a, m\rangle \Downarrow_i v$$

$\langle a, m\rangle \Downarrow_i v$ says that given a **map m**, the **expression a** evaluates to a value $v \in L^i$ (using interval arithmetic)

The operational semantics rules for expression evaluation:

- any constant Z is abstracted to an element in L^i
- operators `+`, `-` and `*` are re-defined for the Interval domain

Arithmetic Expressions

If we add $\perp_i$ to any other element, we get $\perp_i$.

If both operands are not $\perp_i$, we get:

```
[x,y] + [z,q] = [x + z, y + q]
```

what about $*$?

is `[x,y] * [z,q] = [x * z, y * q]` sound ?

Look for all four combinations!

$[\![b]\!]_i$

Let us first look at the expression: $a_1 \; c \; a_2$

Here, c is a condition such as: $\leq$, $=$, $<$

For a memory map m, we need to define : $[\![a_1 \; c \; a_2]\!]_i(m)$ which produces another map as the result.

What is $[\![x \leq y]\!]$?

Easy case: $x_{max} \leq y_{min}$

- We simply keep the intervals of x and y

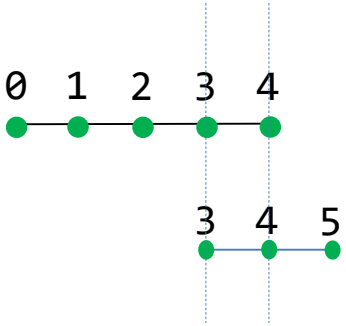
But, suppose we have the program:

```
// Here, x is [0,4] and y is [3,5]
if (x ≤ y){
  1: … // x? y?
}
```

What are the intervals for x and y at label 1 ?

Definition of $[l_1,u_1] \leq [l_2,u_2]$

what should $[0,4] \leq [3,5]$ produce ?



one answer is: $(x=[0,3],y=[3,5])$. Is it sound ?

another non-comparable answer is: $(x=[0,4],y=[4,5])$. Is it sound ?

can you find a more precise answer ?

Definition of $[l_1,u_1] \leq [l_2,u_2]$

$$[l_1,u_1] \leq [l_2,u_2] \;\; = ([l_1,u_1] \sqcap_i [-\infty,u_2],\ [l_1,\infty] \sqcap_i [l_2,u_2])$$

$$[0,4] \leq [3,5] = (x=[0,4] \sqcap_i [-\infty,5], y=[0,\infty] \sqcap_i [3,5])$$
$$= (x=[0,4], y=[3,5])$$

Exercise: define < and =

Evaluating $[\![b]\!]_i$

$$[\![b_1 \vee b_2]\!]_i\,(m) = [\![b_1]\!]_i\,(m) \sqcup [\![b_2]\!]_i\,(m)$$

$$[\![b_1 \wedge b_2]\!]_i\,(m) = [\![b_1]\!]_i\,(m) \sqcap [\![b_2]\!]_i\,(m)$$

F^i on an example

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

What is F^i?

Abstract Interpretation: Step 3

1. select/define an abstract domain
   - selected based on the type of **properties** you want to prove

2. define abstract semantics **for the language** w.r.t. to the domain
   - prove **sound** w.r.t **concrete semantics**
   - involves defining abstract transformers
     - that is, effect of statement / expression on the abstract domain

3. iterate abstract transformers over the abstract domain
   - until we reach a **fixed point**

F^i on an example

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

$F^i(m)1 = \quad \lambda\ v.\ [-\infty,\infty]$

$F^i(m)2 = \quad [\![x := 5]\!]_i\,(m(1))$

$F^i(m)3 = \quad [\![y := 7]\!]_i\,(m(2)) \sqcup [\![goto\ 3]\!]_i\,(m(6))$

$F^i(m)4 = \quad [\![i \geq 0]\!]_i\,(m(3))$

$F^i(m)5 = \quad [\![y := y + 1]\!]_i\,(m(4))$

$F^i(m)6 = \quad [\![i := i - 1]\!]_i\,(m(5))$

$F^i(m)7 = \quad [\![i < 0]\!]_i\,(m(3))$

Fixed point of F^i

Let us compute the least fixed point of F^i

Iterate 0

The collection of these lines denote the current iterate. The iterate is a map

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

2: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

3: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

4: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

5: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

6: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

7: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

Iterate 1

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

3: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

4: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

5: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

6: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

7: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

Iterate 2

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

4: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

5: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

6: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

7: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

Iterate 3

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty,\infty]$

4: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

5: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

6: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

7: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

Iterate 4

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }
7:
}
```

Notice how we propagated to both labels 4 and 7 at the same time

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty,\infty]$

4: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [0,\infty]$

5: $x \rightarrow \bot_i,\ y \rightarrow \bot_i,\ i \rightarrow \bot_i$

6: $x \rightarrow \bot_i,\ y \rightarrow \bot_i,\ i \rightarrow \bot_i$

7: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty, -1]$

Iterate 5

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty,\infty]$

4: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [0,\infty]$

5: $x \rightarrow [5,5],\ y \rightarrow [8,8],\ i \rightarrow [0,\infty]$

6: $x \rightarrow \perp_i,\ y \rightarrow \perp_i,\ i \rightarrow \perp_i$

7: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty, -1]$

Iterate 6

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty,\infty]$

4: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [0,\infty]$

5: $x \rightarrow [5,5],\ y \rightarrow [8,8],\ i \rightarrow [0,\infty]$

6: $x \rightarrow [5,5],\ y \rightarrow [8,8],\ i \rightarrow [-1,\infty]$

7: $x \rightarrow [5,5],\ y \rightarrow [7,7],\ i \rightarrow [-\infty, -1]$

Iterate 7

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,8], i → [-∞,∞]

4: x → [5,5], y → [7,7], i → [0,∞]

5: x → [5,5], y → [8,8], i → [0,∞]

6: x → [5,5], y → [8,8], i → [-1,∞]

7: x → [5,5], y → [7,7], i → [-∞, -1]

Iterate 8

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,8], i → [-∞,∞]

4: x → [5,5], y → [7,8], i → [0,∞]

5: x → [5,5], y → [8,8], i → [0,∞]

6: x → [5,5], y → [8,8], i → [-1,∞]

7: x → [5,5], y → [7,8], i → [-∞, -1]

Iterate 9

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,8], i → [-∞,∞]

4: x → [5,5], y → [7,8], i → [0,∞]

5: x → [5,5], y → [8,9], i → [0,∞]

6: x → [5,5], y → [8,8], i → [-1,∞]

7: x → [5,5], y → [7,8], i → [-∞, -1]

Iterate 10

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,8], i → [-∞,∞]

4: x → [5,5], y → [7,8], i → [0,∞]

5: x → [5,5], y → [8,9], i → [0,∞]

6: x → [5,5], y → [8,9], i → [-1,∞]

7: x → [5,5], y → [7,8], i → [-∞, -1]

Iterate 11

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,9], i → [-∞,∞]

4: x → [5,5], y → [7,8], i → [0,∞]

5: x → [5,5], y → [8,9], i → [0,∞]

6: x → [5,5], y → [8,9], i → [-1,∞]

7: x → [5,5], y → [7,8], i → [-∞, -1]

Iterate 12

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,9], i → [-∞,∞]

4: x → [5,5], y → [7,9], i → [0,∞]

5: x → [5,5], y → [8,9], i → [0,∞]

6: x → [5,5], y → [8,9], i → [-1,∞]

7: x → [5,5], y → [7,9], i → [-∞, -1]

Iterate 13

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,9], i → [-∞,∞]

4: x → [5,5], y → [7,9], i → [0,∞]

5: x → [5,5], y → [8,10], i → [0,∞]

6: x → [5,5], y → [8,9], i → [-1,∞]

7: x → [5,5], y → [7,9], i → [-∞, -1]

Iterate 14

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5],\ y \rightarrow [-\infty,\infty],\ i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5],\ y \rightarrow [7,9],\ i \rightarrow [-\infty,\infty]$

4: $x \rightarrow [5,5],\ y \rightarrow [7,9],\ i \rightarrow [0,\infty]$

5: $x \rightarrow [5,5],\ y \rightarrow [8,10],\ i \rightarrow [0,\infty]$

6: $x \rightarrow [5,5],\ y \rightarrow [8,10],\ i \rightarrow [-1,\infty]$

7: $x \rightarrow [5,5],\ y \rightarrow [7,9],\ i \rightarrow [-\infty, -1]$

Iterate 15

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: $x \rightarrow [-\infty,\infty]$, $y \rightarrow [-\infty,\infty]$, $i \rightarrow [-\infty,\infty]$

2: $x \rightarrow [5,5]$, $y \rightarrow [-\infty,\infty]$, $i \rightarrow [-\infty,\infty]$

3: $x \rightarrow [5,5]$, $y \rightarrow [7,10]$, $i \rightarrow [-\infty,\infty]$

4: $x \rightarrow [5,5]$, $y \rightarrow [7,9]$, $i \rightarrow [0,\infty]$

5: $x \rightarrow [5,5]$, $y \rightarrow [8,10]$, $i \rightarrow [0,\infty]$

6: $x \rightarrow [5,5]$, $y \rightarrow [8,10]$, $i \rightarrow [-1,\infty]$

7: $x \rightarrow [5,5]$, $y \rightarrow [7,9]$, $i \rightarrow [-\infty, -1]$

Iterate 16

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }
7:
}
```

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]

2: x → [5,5], y → [-∞,∞], i → [-∞,∞]

3: x → [5,5], y → [7,10], i → [-∞,∞]

4: x → [5,5], y → [7,10], i → [0,∞]

5: x → [5,5], y → [8,10], i → [0,∞]

6: x → [5,5], y → [8,10], i → [-1,∞]

7: x → [5,5], y → [7,10], i → [-∞, -1]

The issue is that the iterates:

$$F^{i\ (0)}\ (\lambda\ \ell\ .\ \lambda\ v.\ \bot_i\ ),\ F^{i\ (1)}\ (\lambda\ \ell\ .\ \lambda\ v.\ \bot_i\ ),\ F^{i\ (2)}\ (\lambda\ \ell\ .\ \lambda\ v.\ \bot_i\ ),\ \ldots$$

will keep going on forever as the value of variable **y** will keep increasing. Hence, we will not be able to compute all of the iterates that we need in order to apply the fixed point theorem.

what should we do in this case ?

Generally, if we have a complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ and a monotone function F, if the height is infinite or the computation of the iterates of F takes too long, we need to find a way to **approximate** the least fixed point of F.

The interval domain and its function F^i is an example of this case.

We need to find a way to compute an element A such that:

$$\mathrm{lfp}^{\sqsubseteq} F \sqsubseteq A$$

Widening Operator

The operator $\nabla: L \times L \rightarrow L$ is called a widening operator if:

- $\forall\ a,b \in L: a \sqcup b \sqsubseteq a \nabla b$ (widening approximates the join)
- if $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \ldots \sqsubseteq x^n \sqsubseteq \ldots$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \ldots \sqsubseteq y^n$ stabilizes after a finite number of steps

  where $y^0 = x^0$ and $\forall i \geq 0$: $y^{i+1} = y^i \nabla x^{i+1}$

Widening is completely **independent of the function F.**

Much like the join, it is an operator defined for the particular domain.