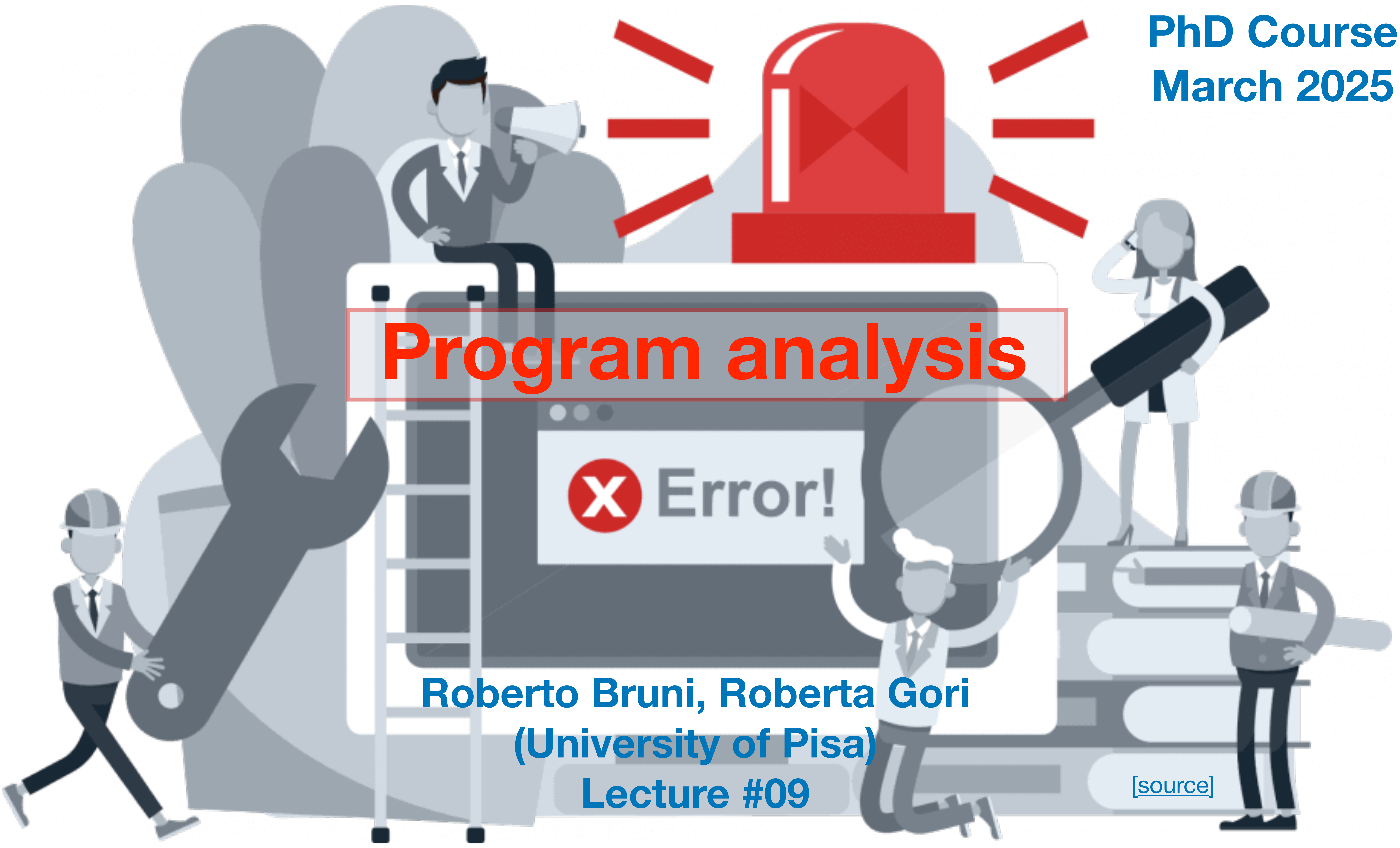
PhD Course
March 2025

Program analysis



Roberto Bruni, Roberta Gori
(University of Pisa)

Lecture #09

[source]

Abstract Interpretation

Abstract Interpretation

It is a technique to formally reason on approximations

It allows to derive effective methods to compute approximations

Generally used to compute overapproximations

Seldom used to compute underapproximations

Example: out of bounds

```
function arrayOutOfBounds(int n, int x[10]) {
  a = 0
  if n >= 10 then
     n = n - 5
  else
     a = ++n

  a = max(0,a — n)
  return x[a] }
```

Let us assume $n \geq 0$

Is it a safe access? ($0 \leq a \leq 9$?)

Using exact semantics

```
function arrayOutOfBounds(int n, int x[10]) {
(0,_)(1,_)(2,_)(3,_)(4,_)(5,_)(6,_)(7,_)(8,_)(9,_)(10,_)….
  a = 0
(0,0)(1,0)(2,0)(3,0)(4,0)(5,0)(6,0)(7,0)(8,0)(9,0)(10,0)….
  if n >= 10 then
(10,0)(11,0)(12,0)(13,0)(14,0)(15,0)(16,0)(17,0)(18,0)(19,0)….
     n = n - 5
(5,0)(6,0)(7,0)(8,0)(9,0)(10,0)(11,0)(12,0)(13,0)(14,0)….
  else

     a = ++n


  a = max(0,a – n)

  return x[a] }
```

We can't track the infinite set of pairs!

use intervals !

Example: interval abstraction

```
 function arrayOutOfBounds(int n, int x[10]) {
[0,∞]
    a = 0
[0,∞][0,0]
    if n >= 10 then
     [10,∞][0,0]
       n = n - 5
     [5,∞][0,0]
    else
     [0,9][0,0]
       a = ++n
     [1,10][1,10]
[1,∞][0,10]
    a = max(0,a – n)
[1,∞][0,9]
    return x[a] }
```

Merging branches looses precision

safe! $0 \leq a \leq 9$!

Abstract Interpretation: the idea

Goal: Compute the set S of possible values at each line of code

But… this is not feasible in general

We want to find an (over)approximation $S \subseteq S^{\#}$

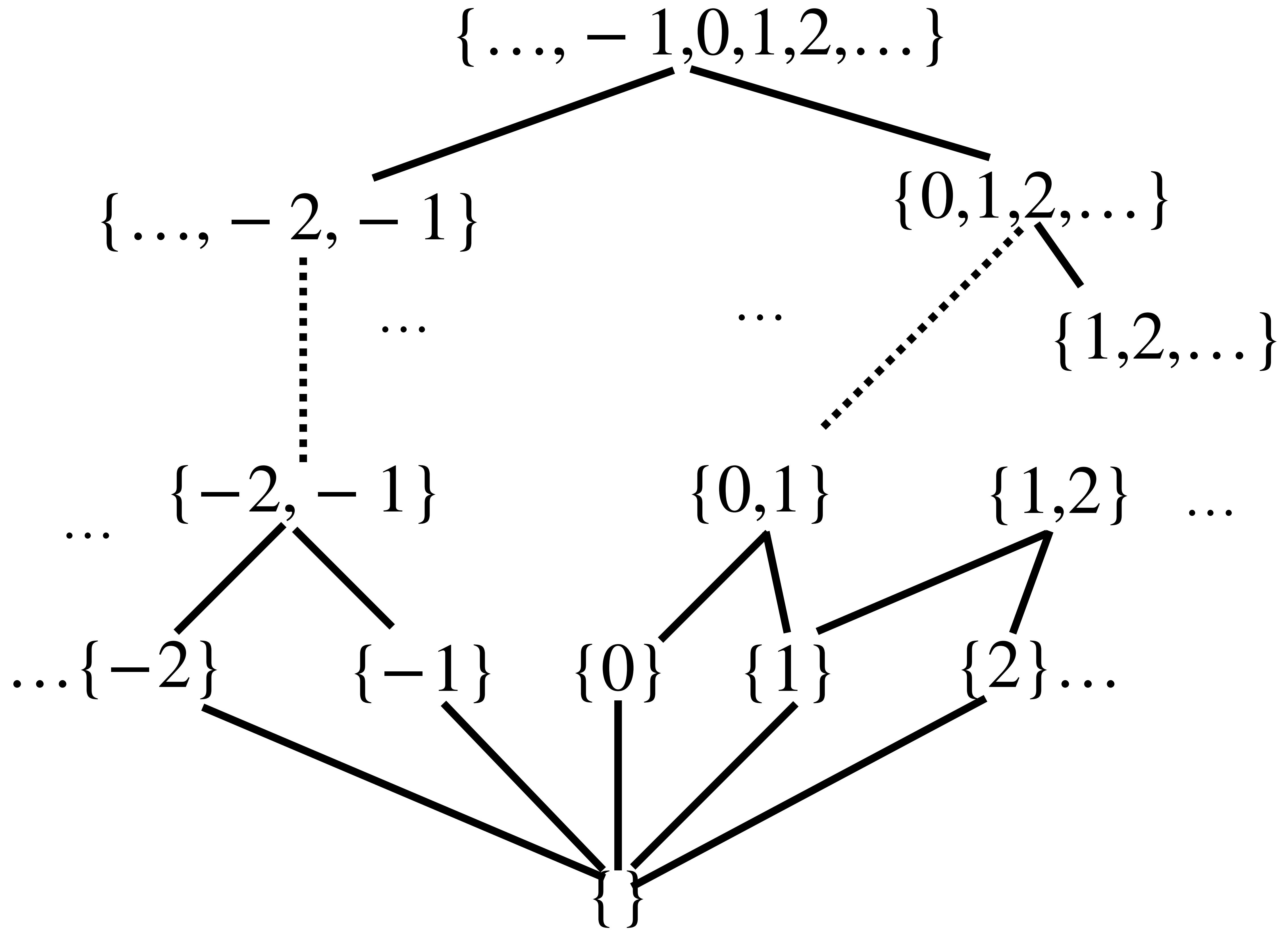
The theory of abstract interpretation allows to compute $S^{\#}$ as a set of abstract values obtained by applying abstract operations

Abstraction and concretization

Concrete domain

The set of values S that we would like to compute belongs to the concrete domain C

$$(\wp(\mathbb{Z}), \subseteq)$$

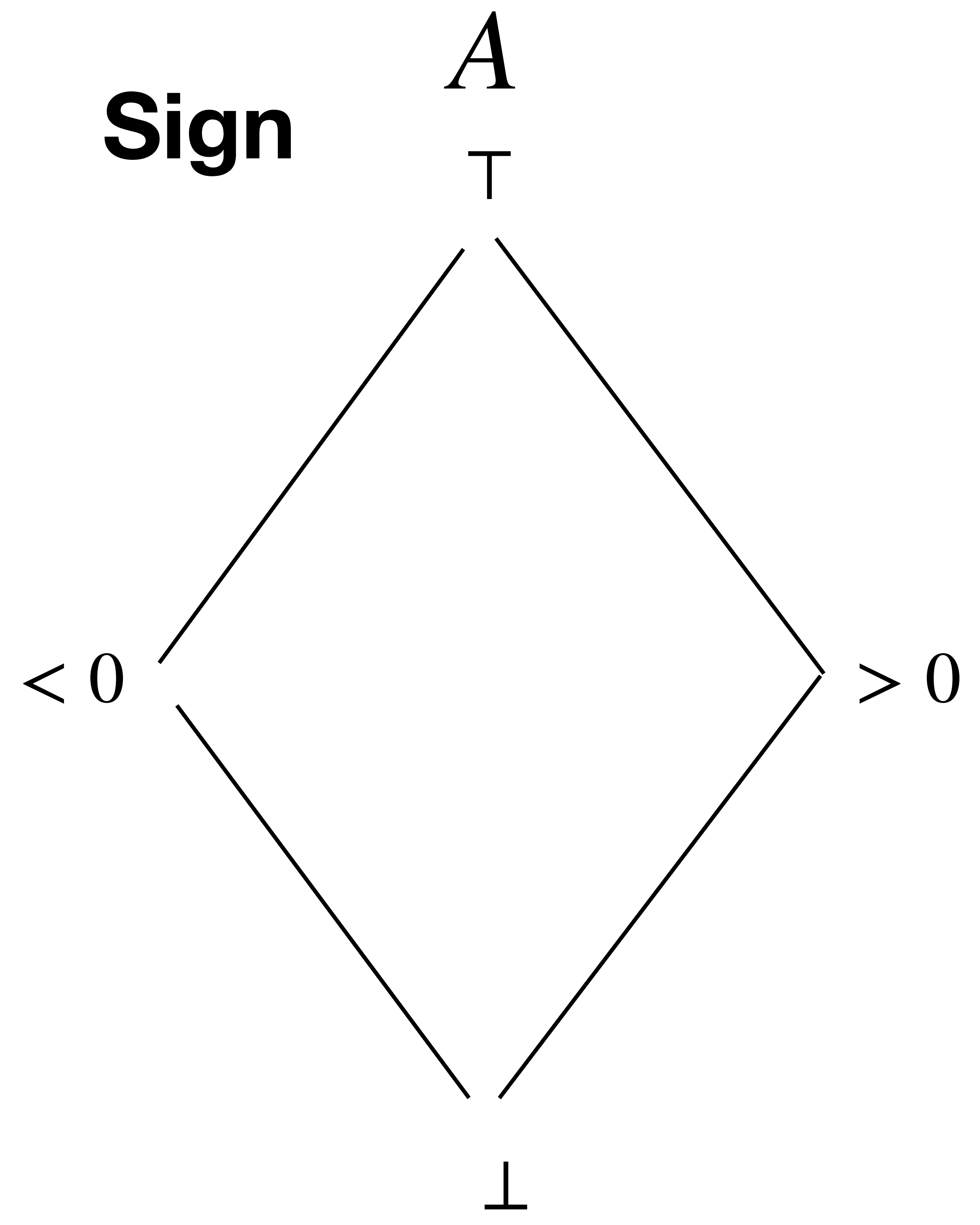


Abstract Domain

$(A, \sqsubseteq)$ expresses some properties of the concrete values

For example

Sign



The order $\sqsubseteq$ on the abstract domain reflects the precision

e.g $\bot \sqsubseteq <$

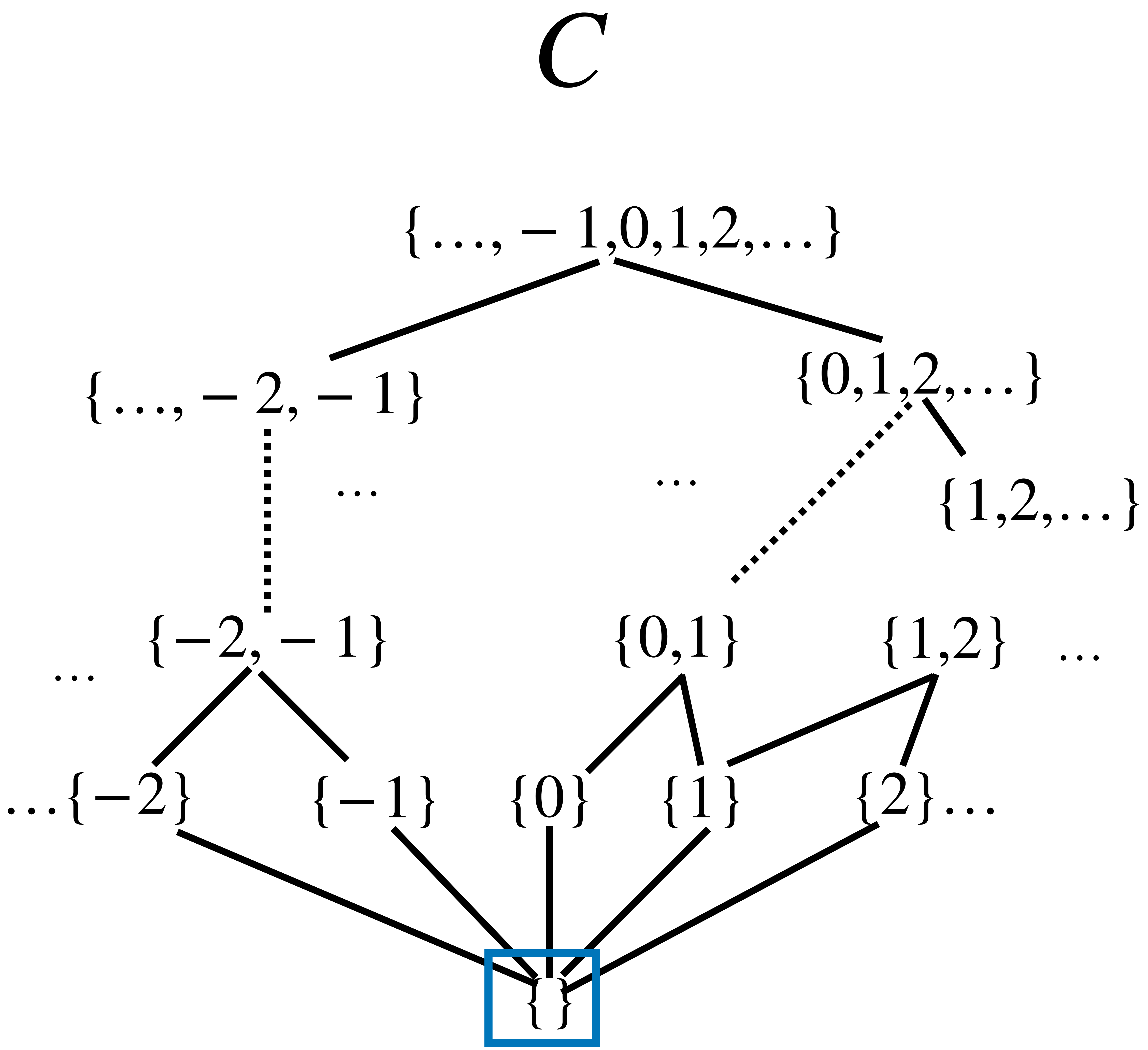
Ingredients of Abstract Interpretation

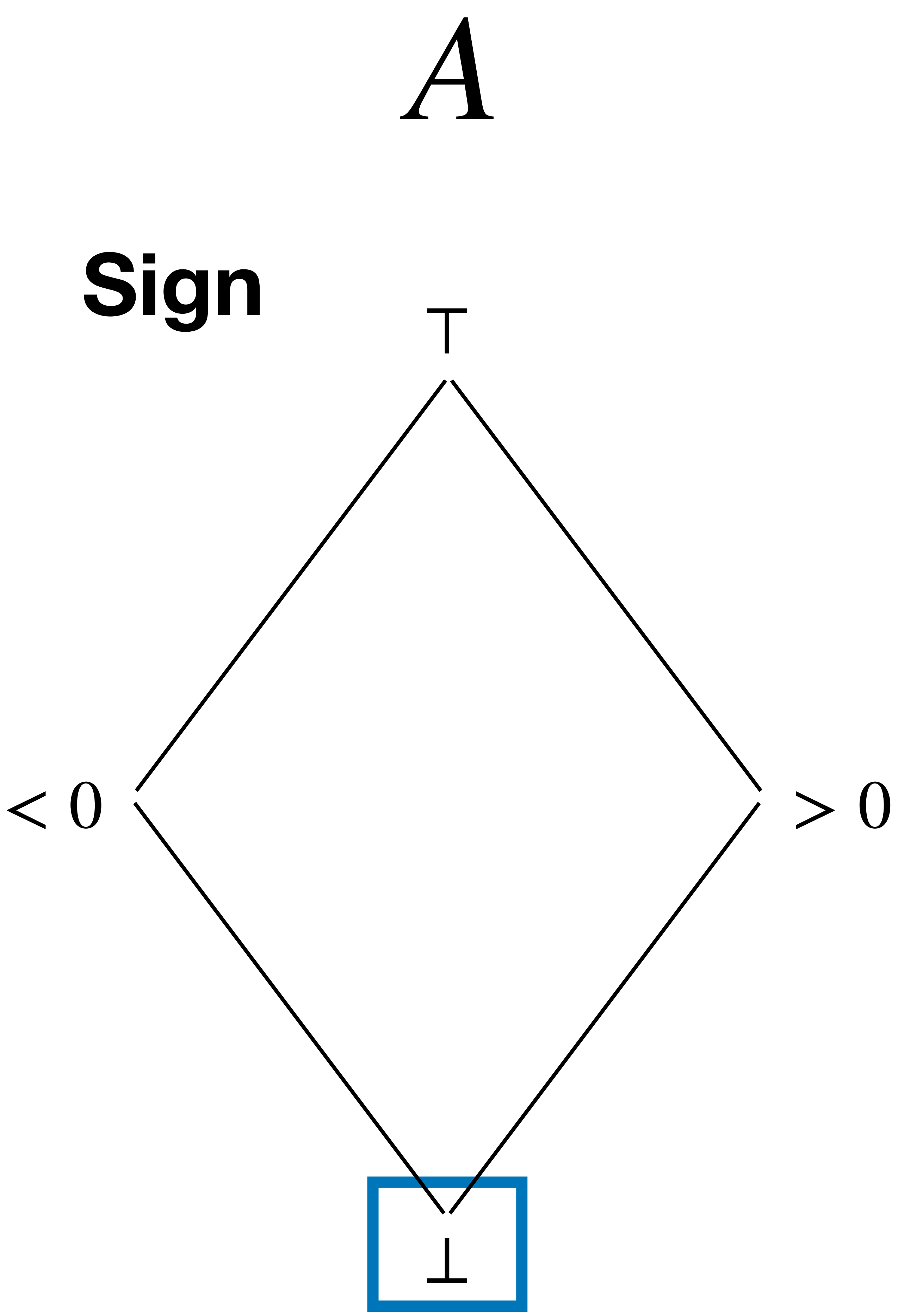
- A concrete domain C

- An abstract domain A

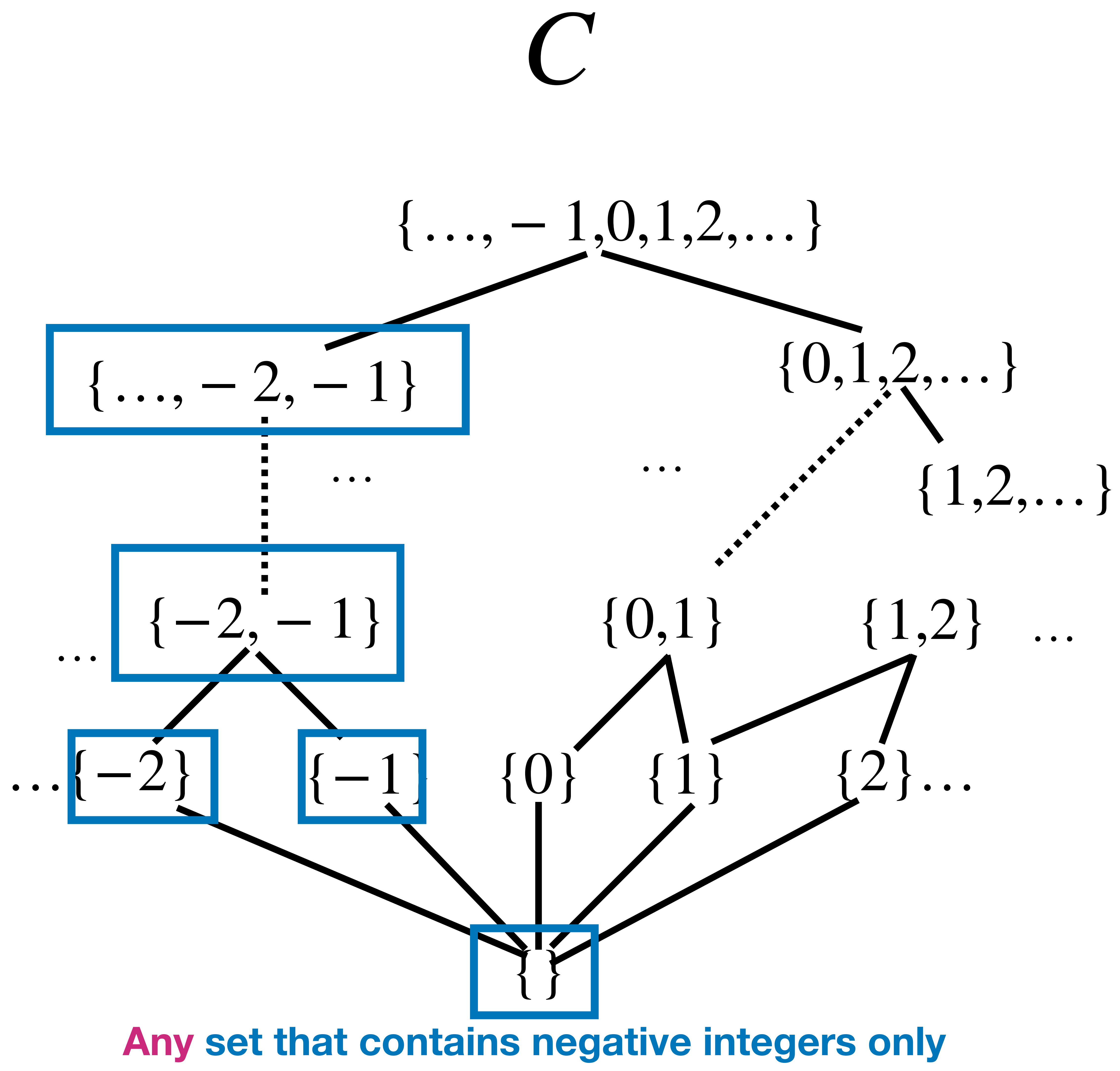
- An abstraction function α that connects the concrete domain to the abstract one
- A concretisation function γ that relates the abstract domain to the concrete one

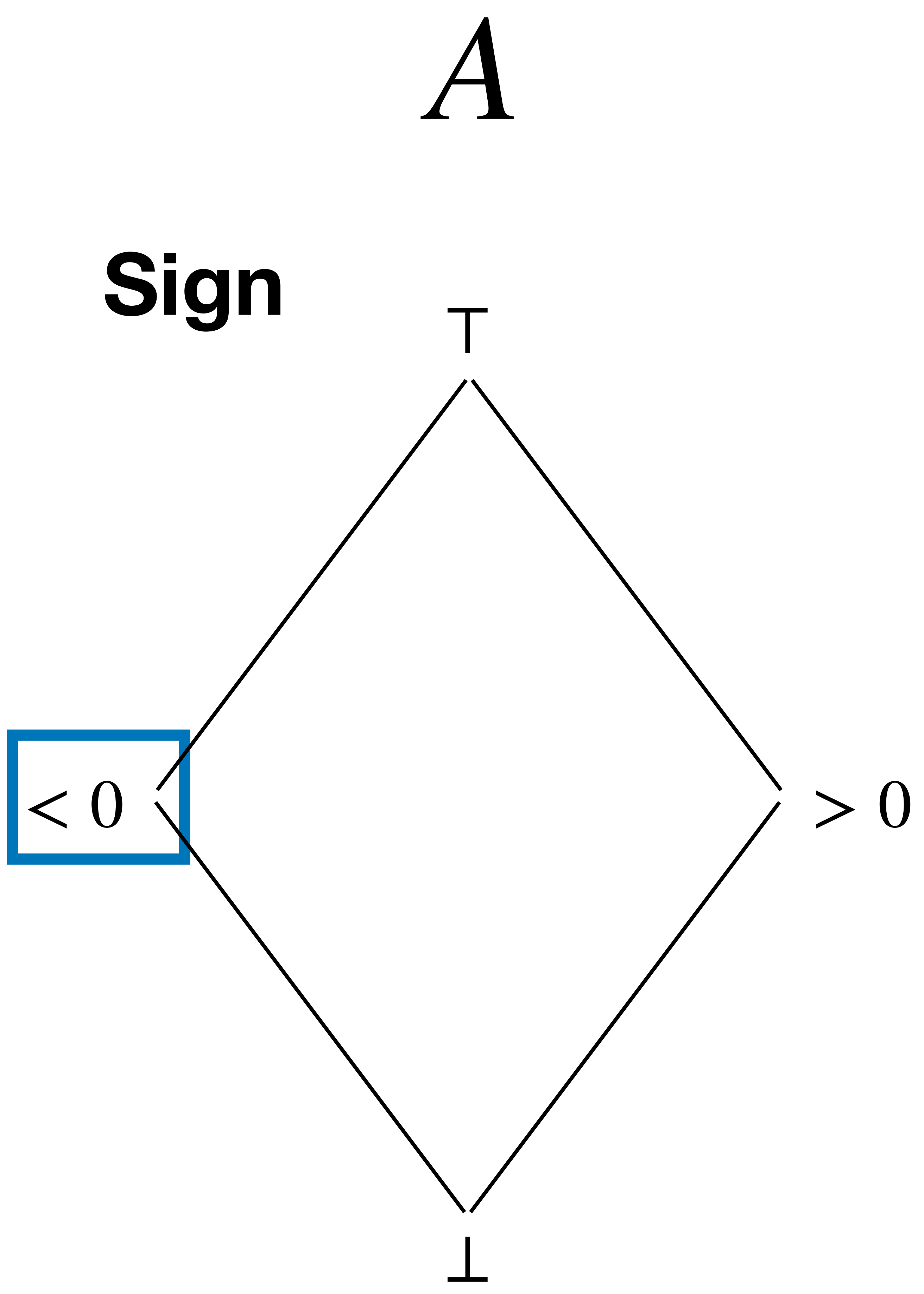
Defining concretisation



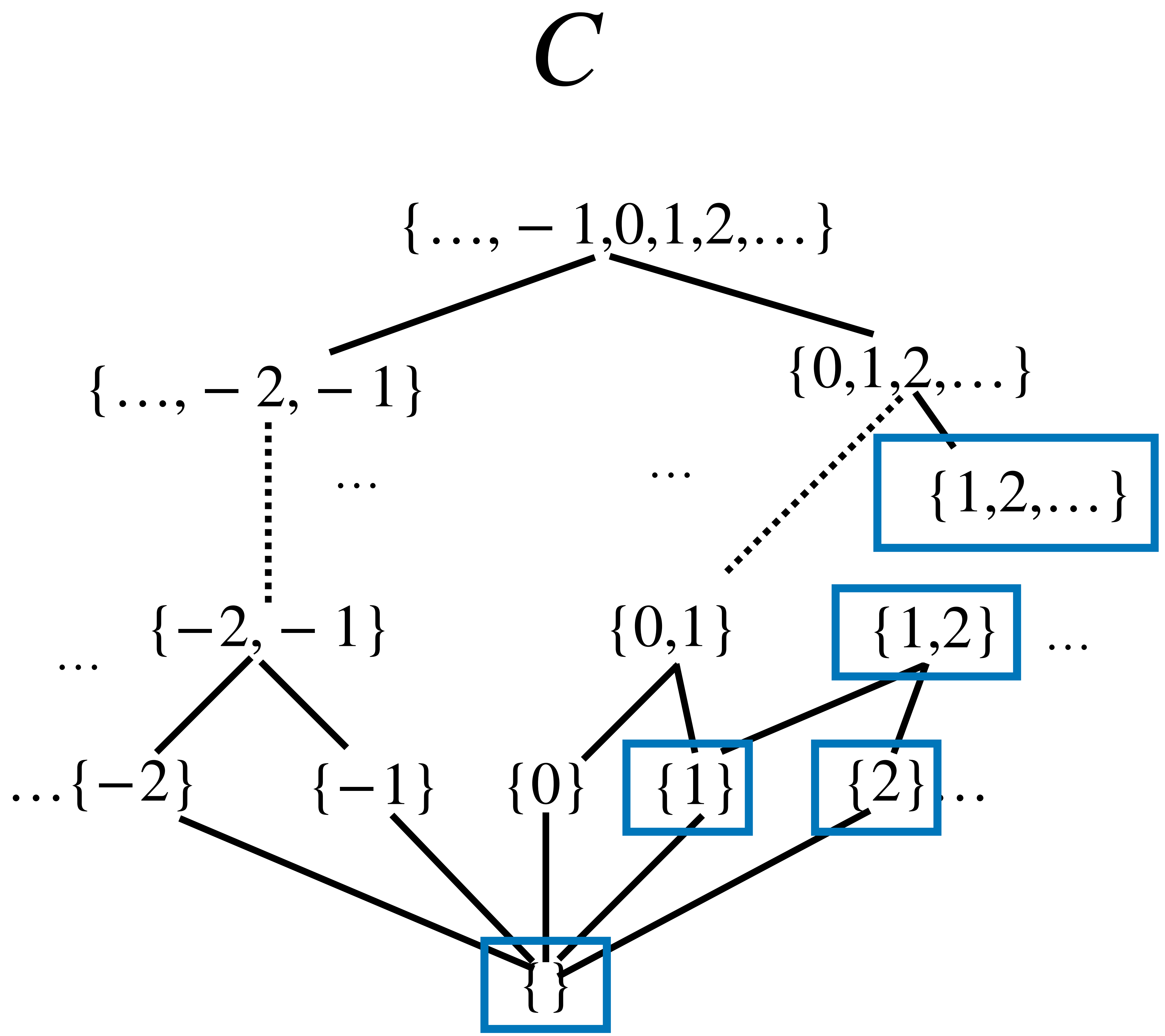


Defining approximation

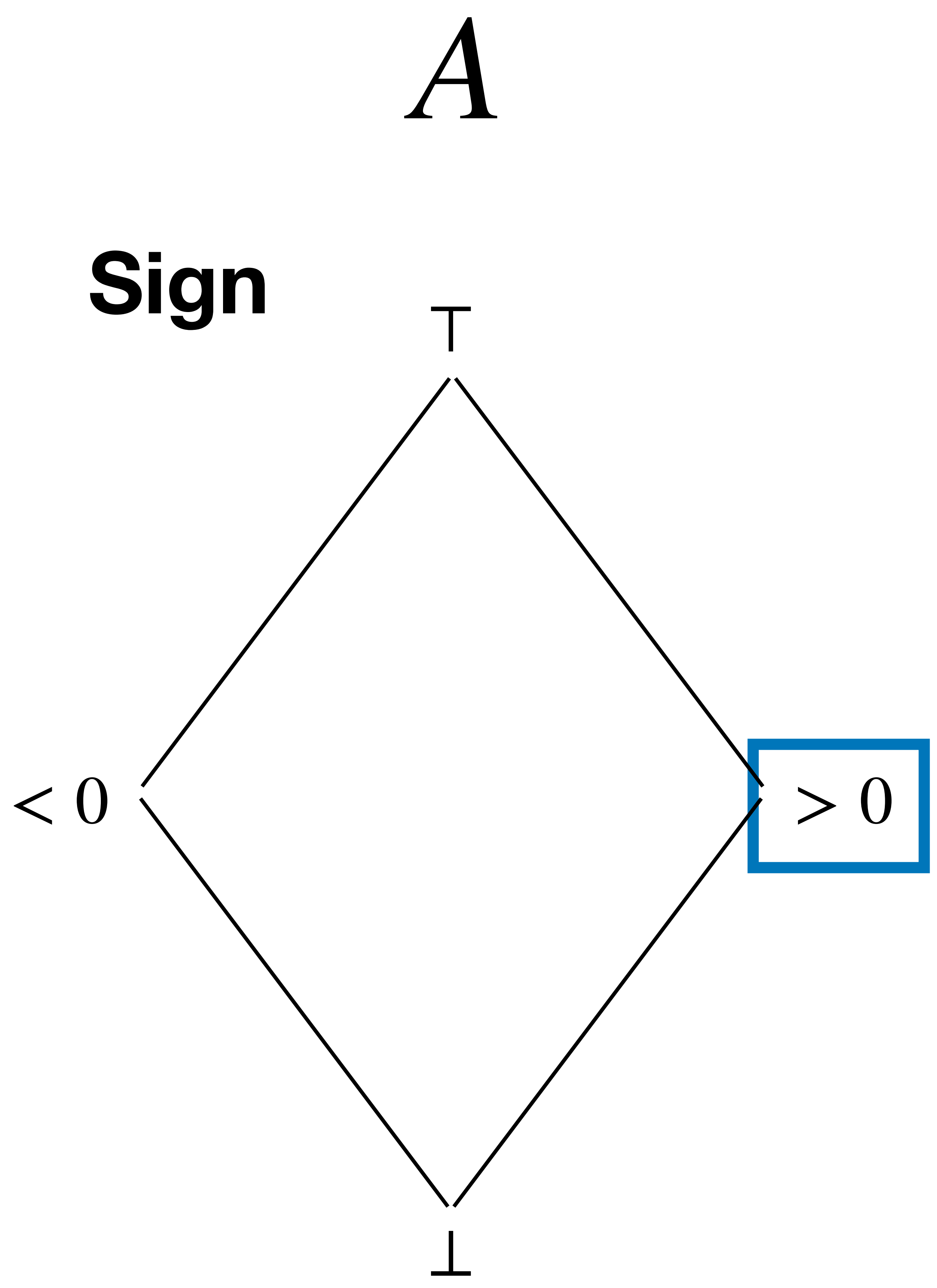




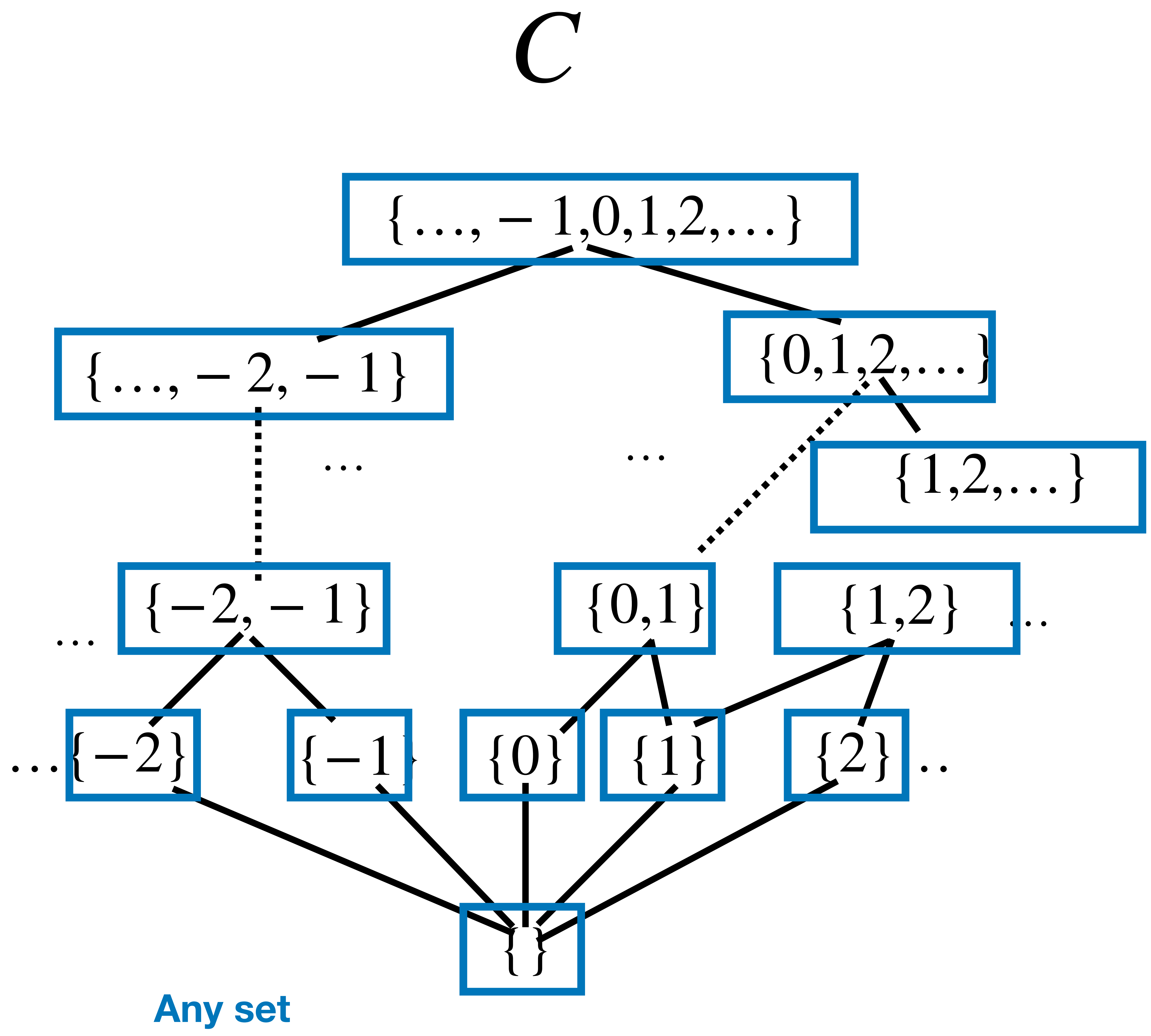
Defining approximation

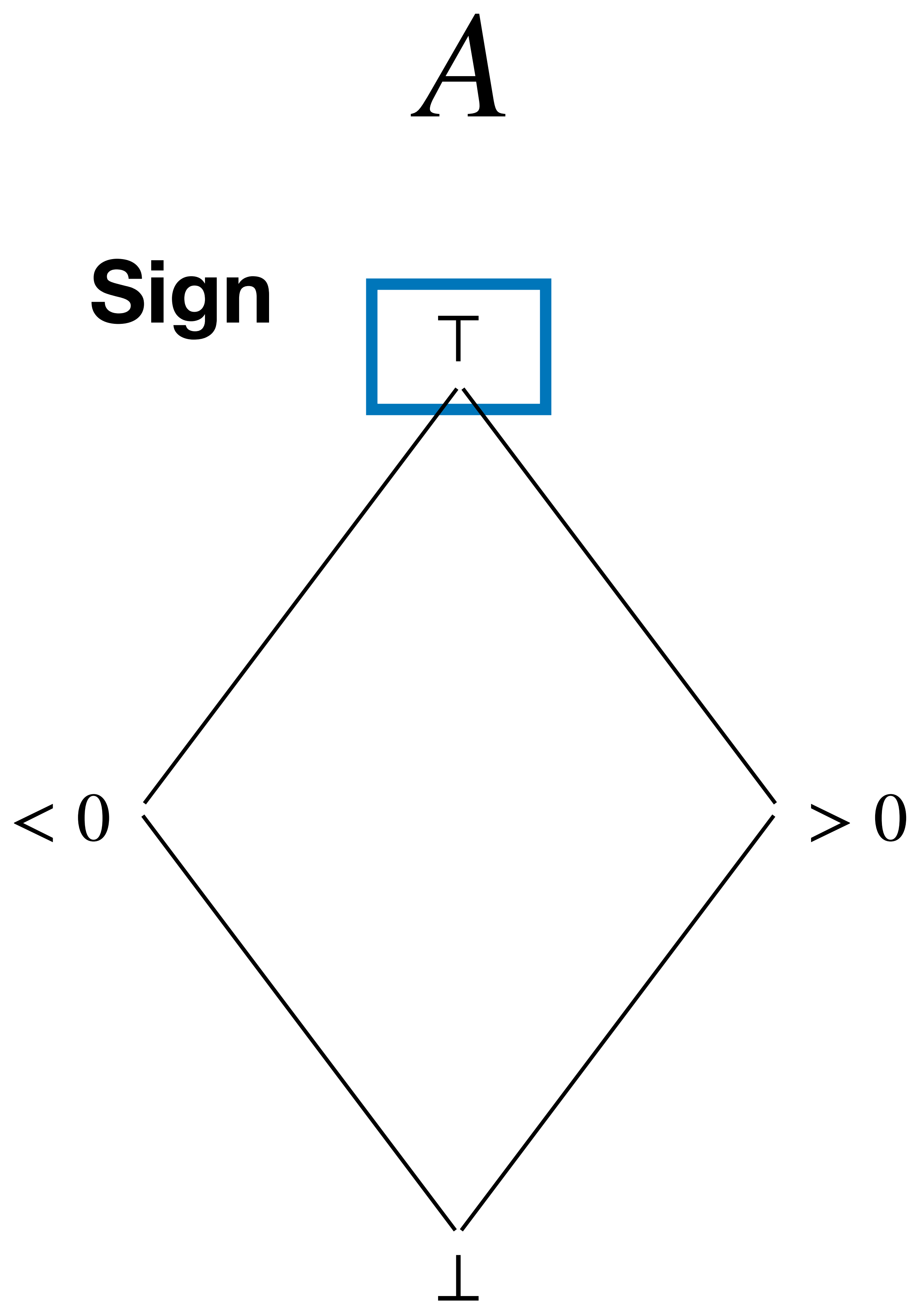


Any set that contains positive integers only



Defining approximation

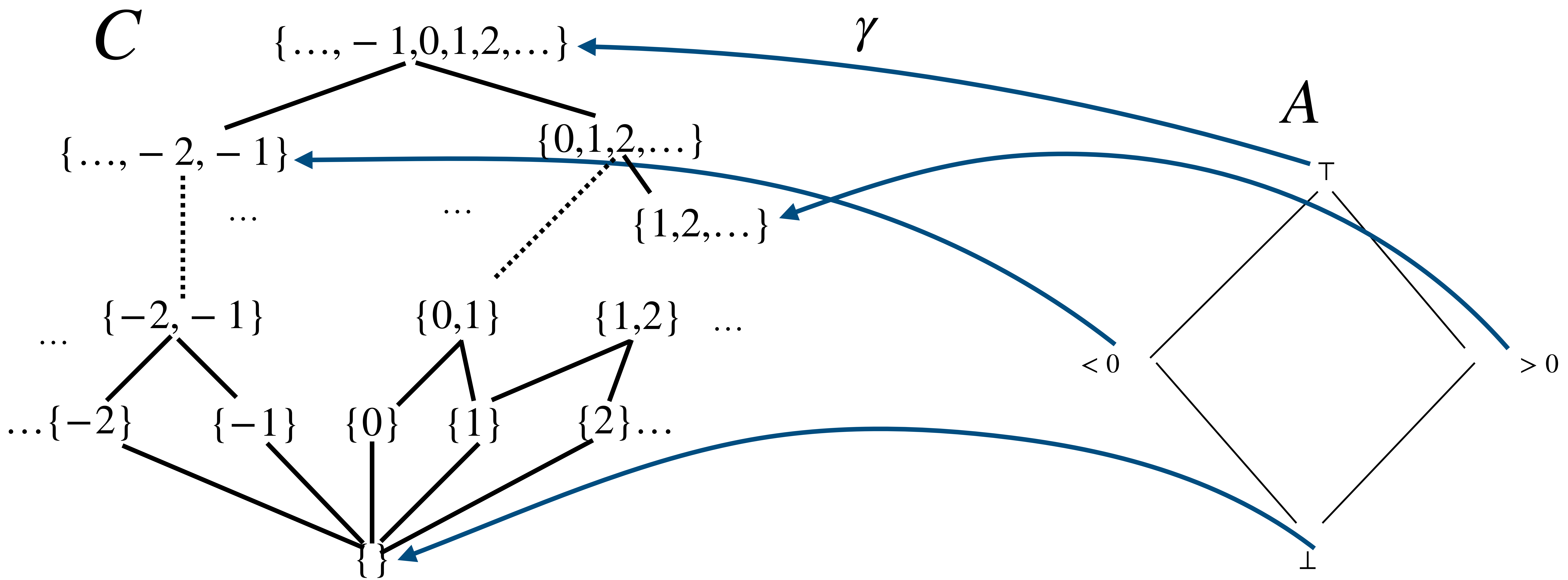




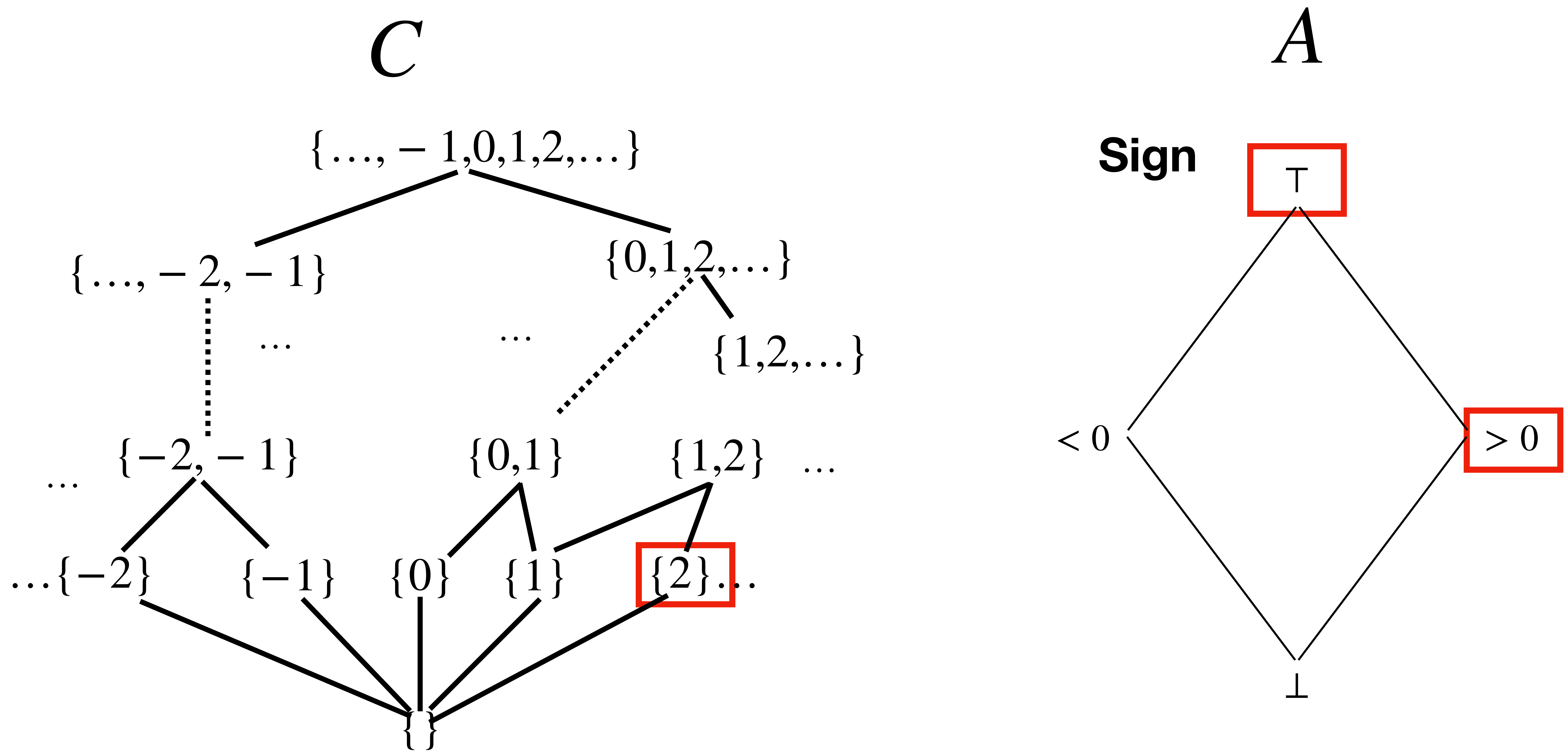
Concretization function

Definition

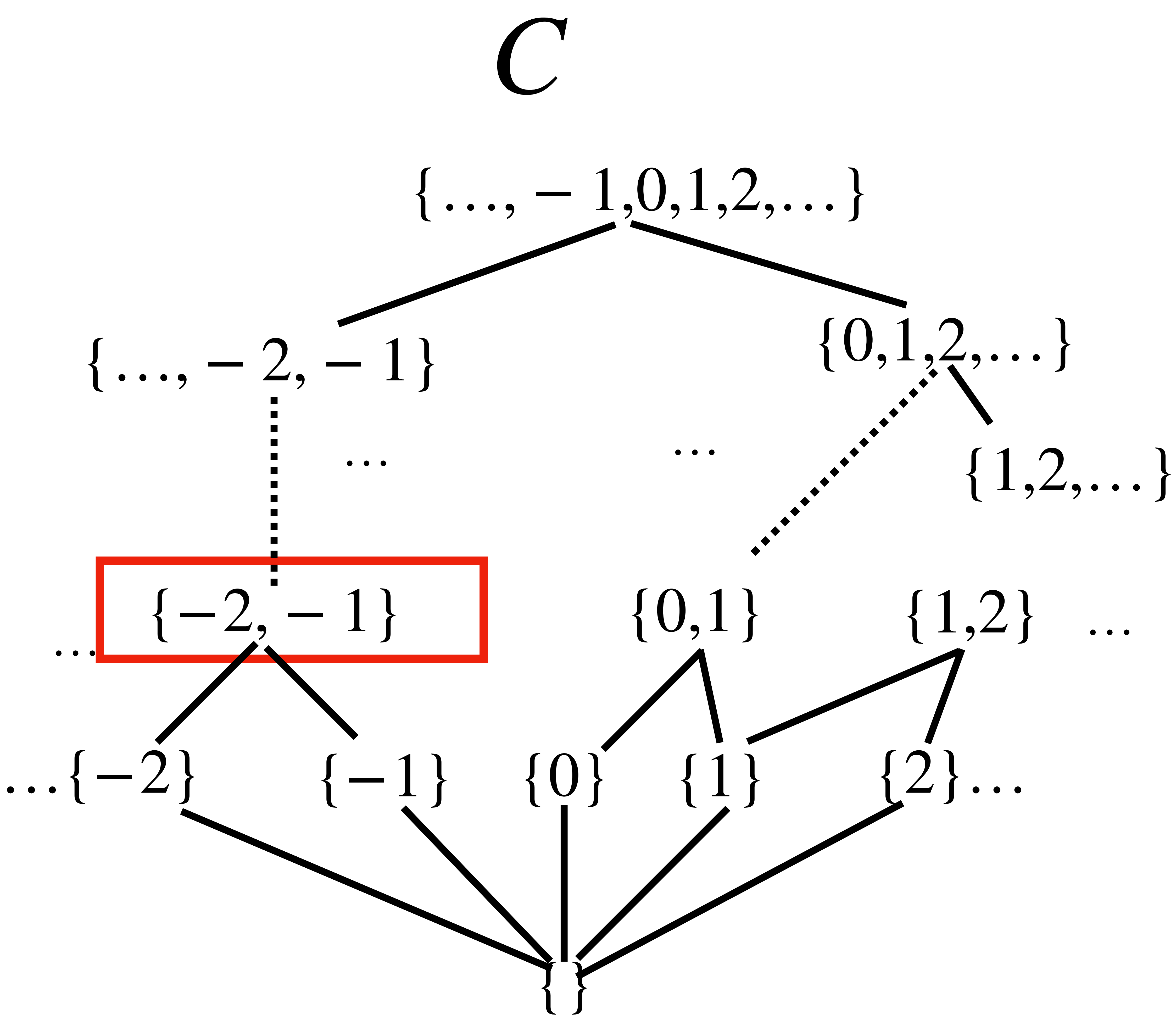
Concretization function $\gamma : A \to C$ is a monotone function that maps abstract a into the greatest concrete c that it approximates

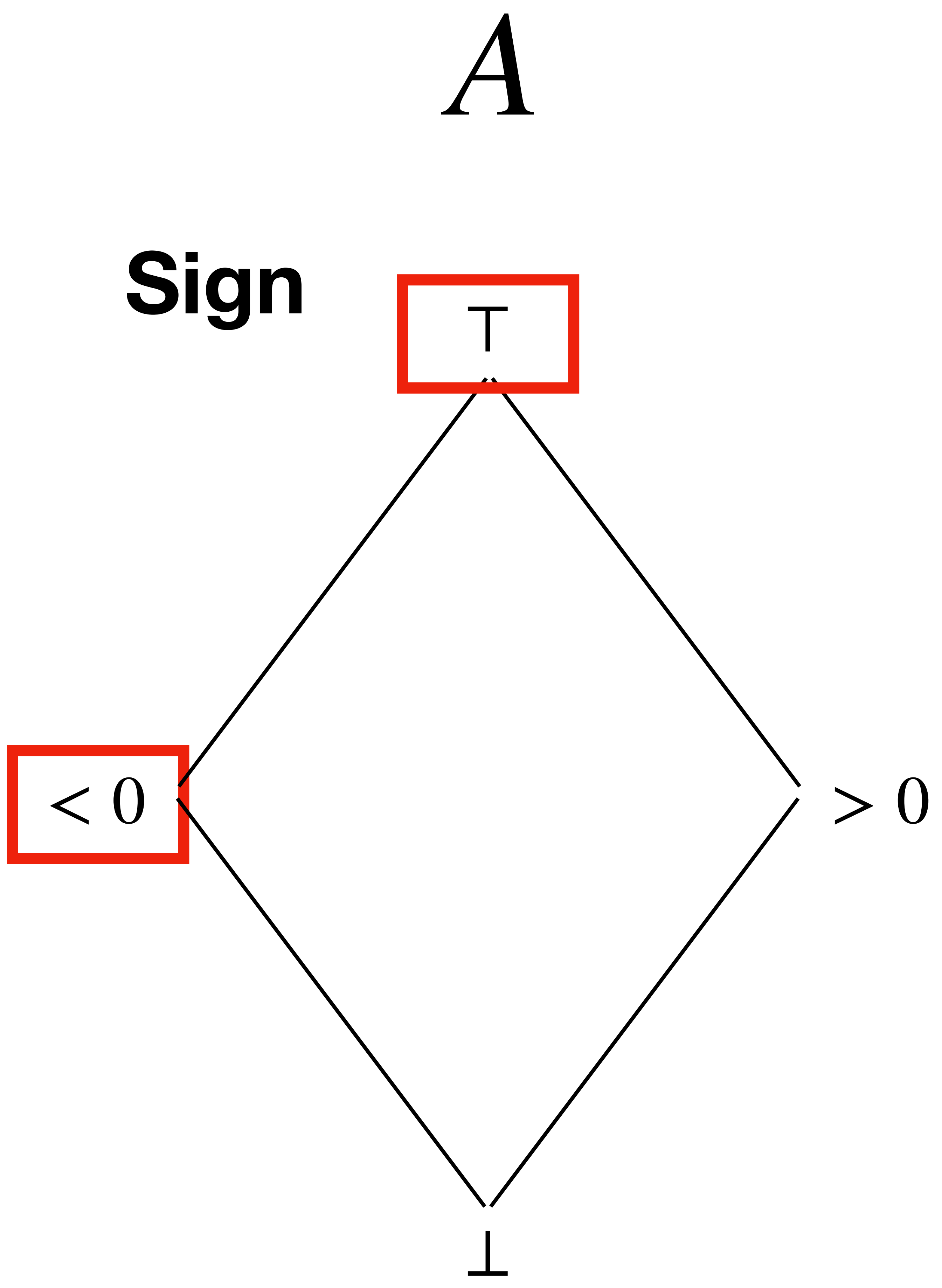


Defining approximation

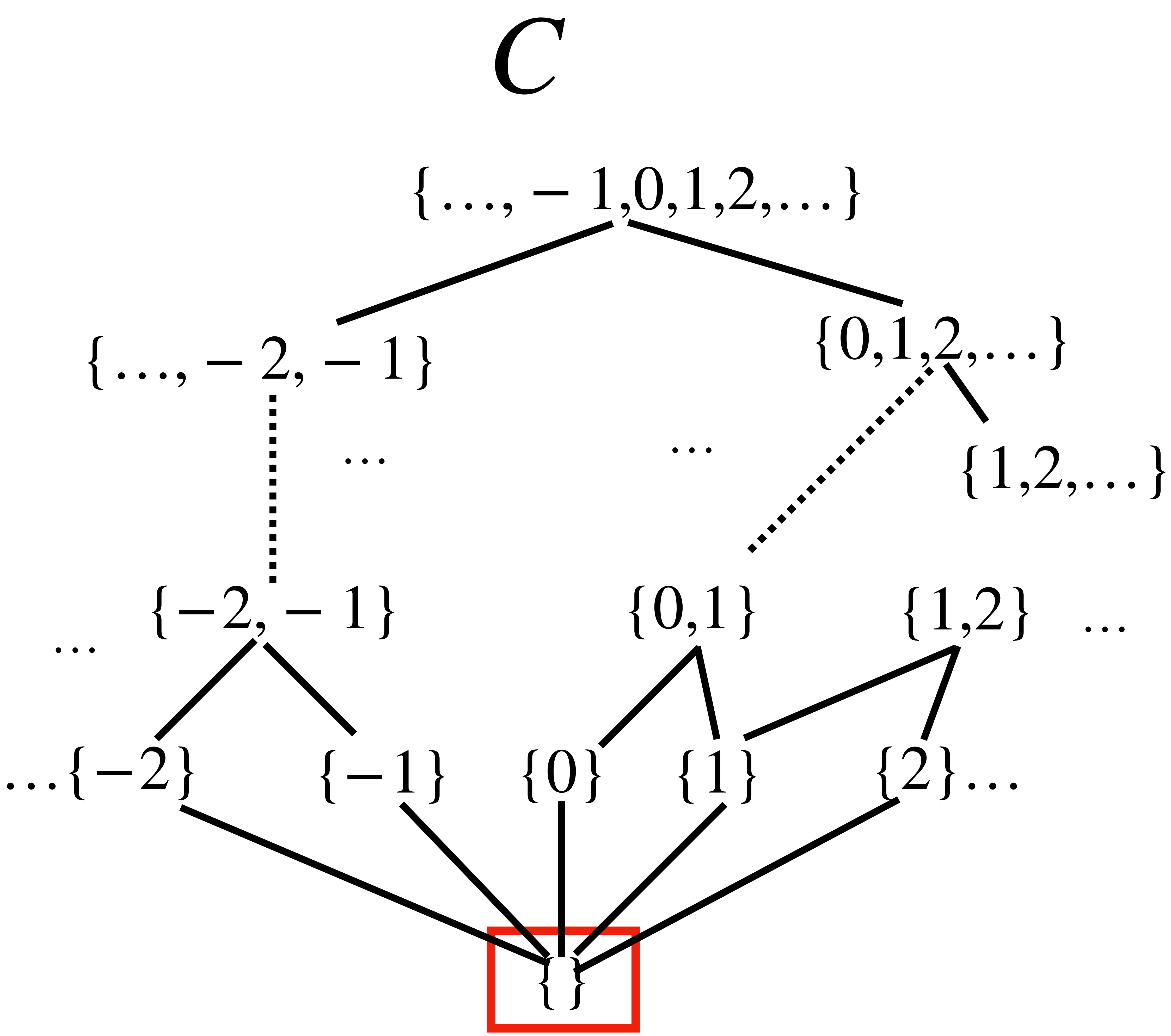


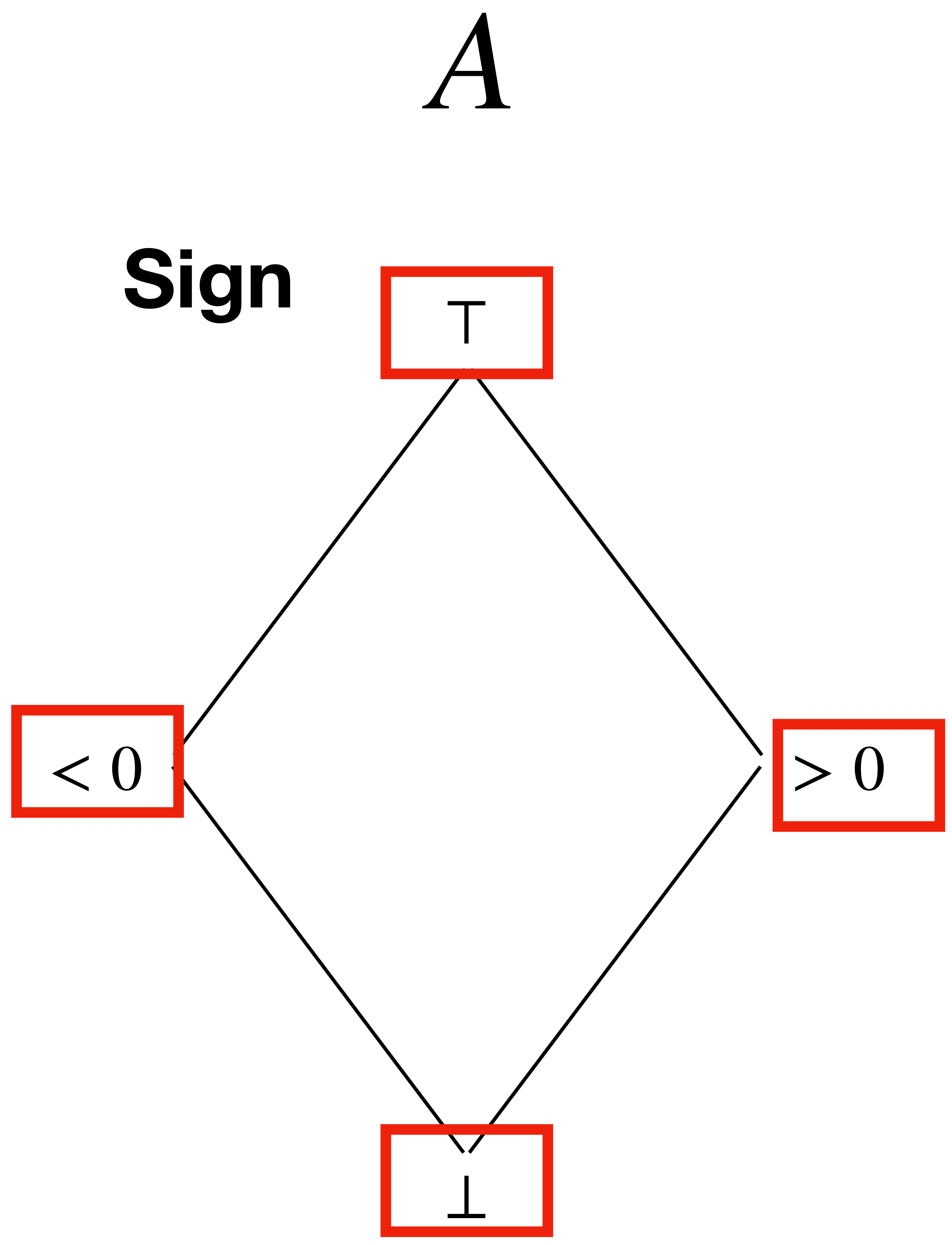
Defining approximation



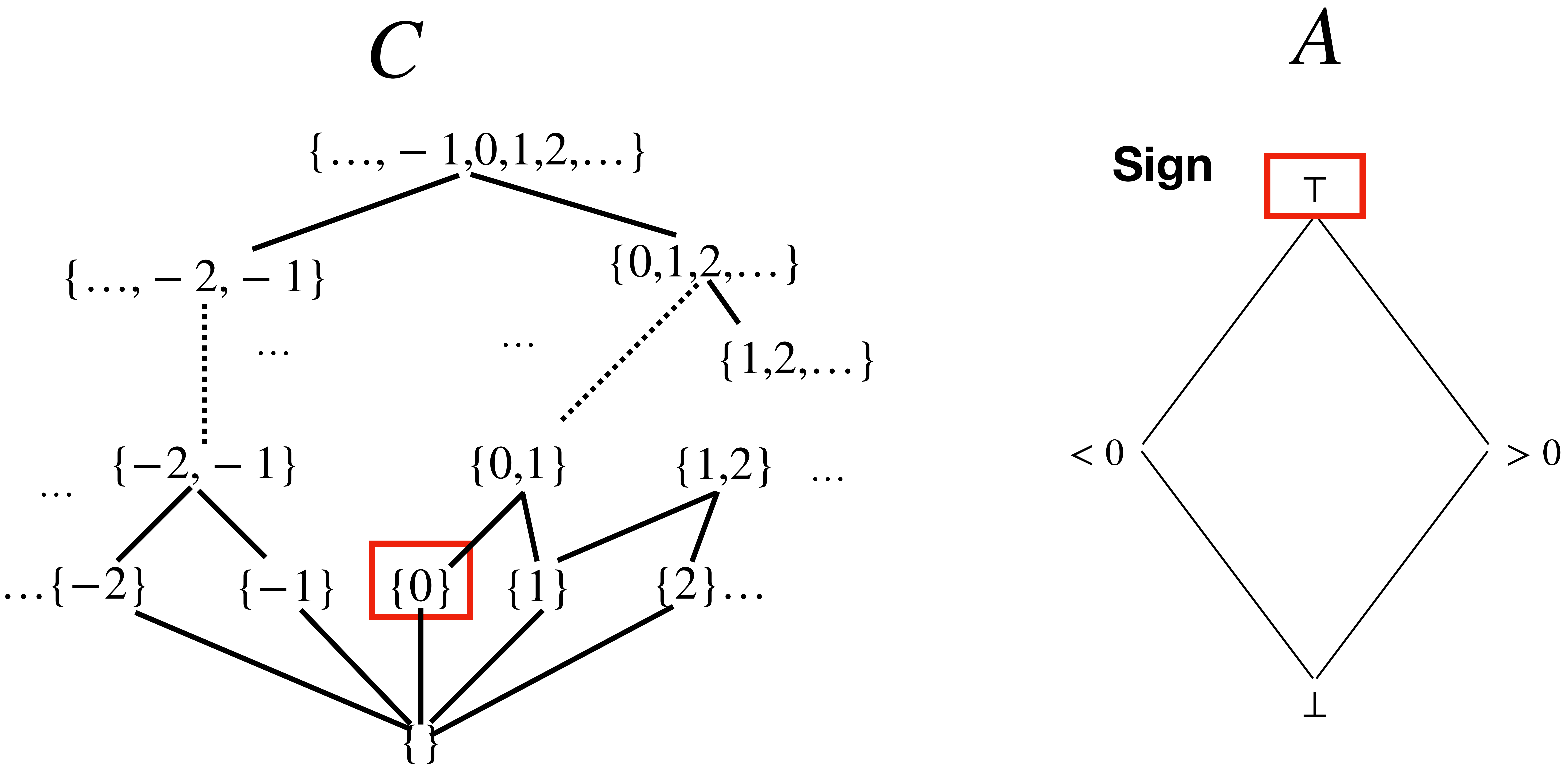


Defining approximation





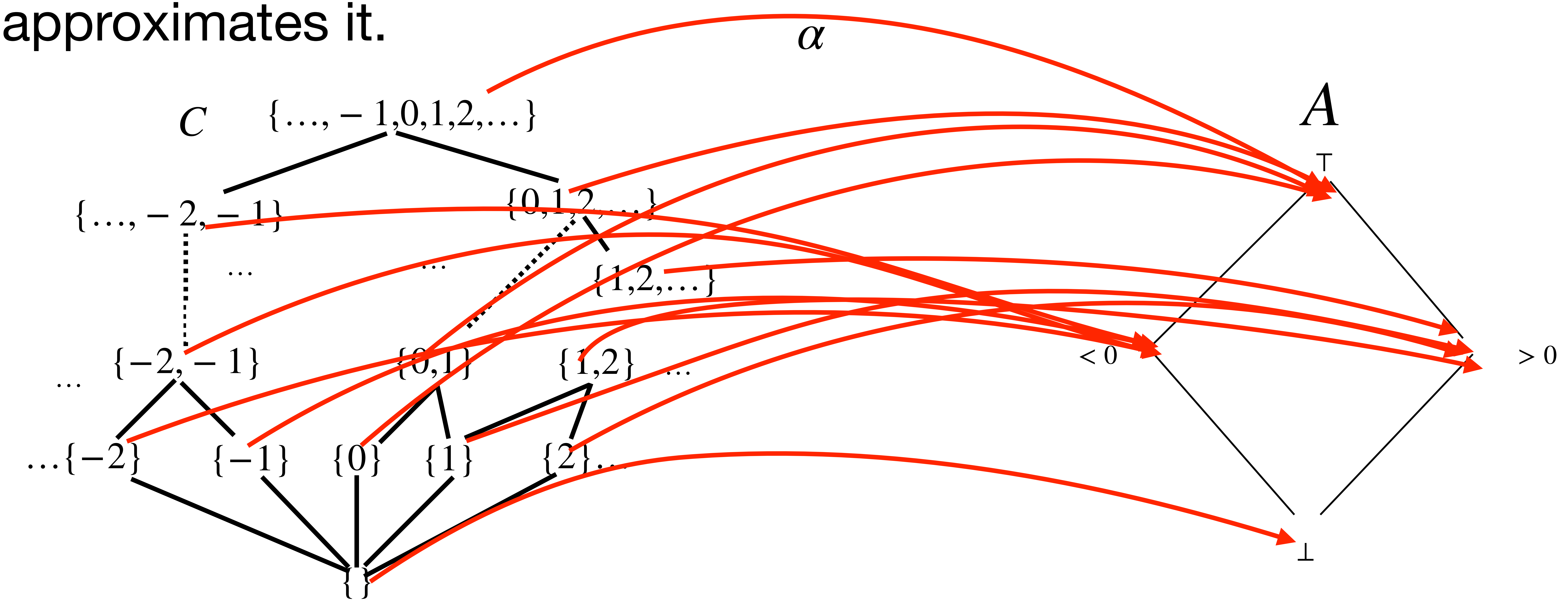
Defining approximation



Abstraction function

Definition

Abstraction function $\alpha : C \to A$ is a monotone function that maps concrete c into the most precise abstract element that approximates it.

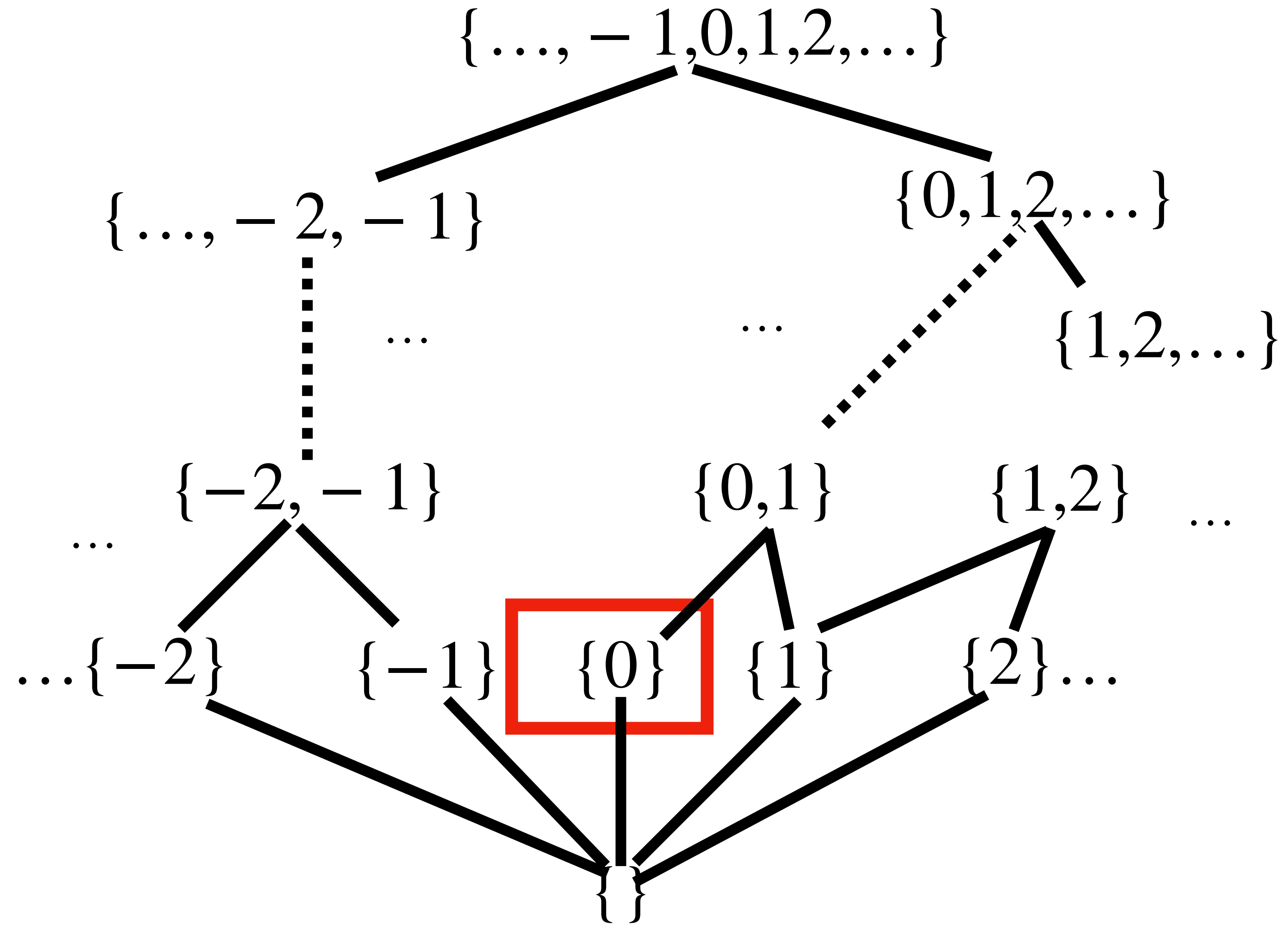


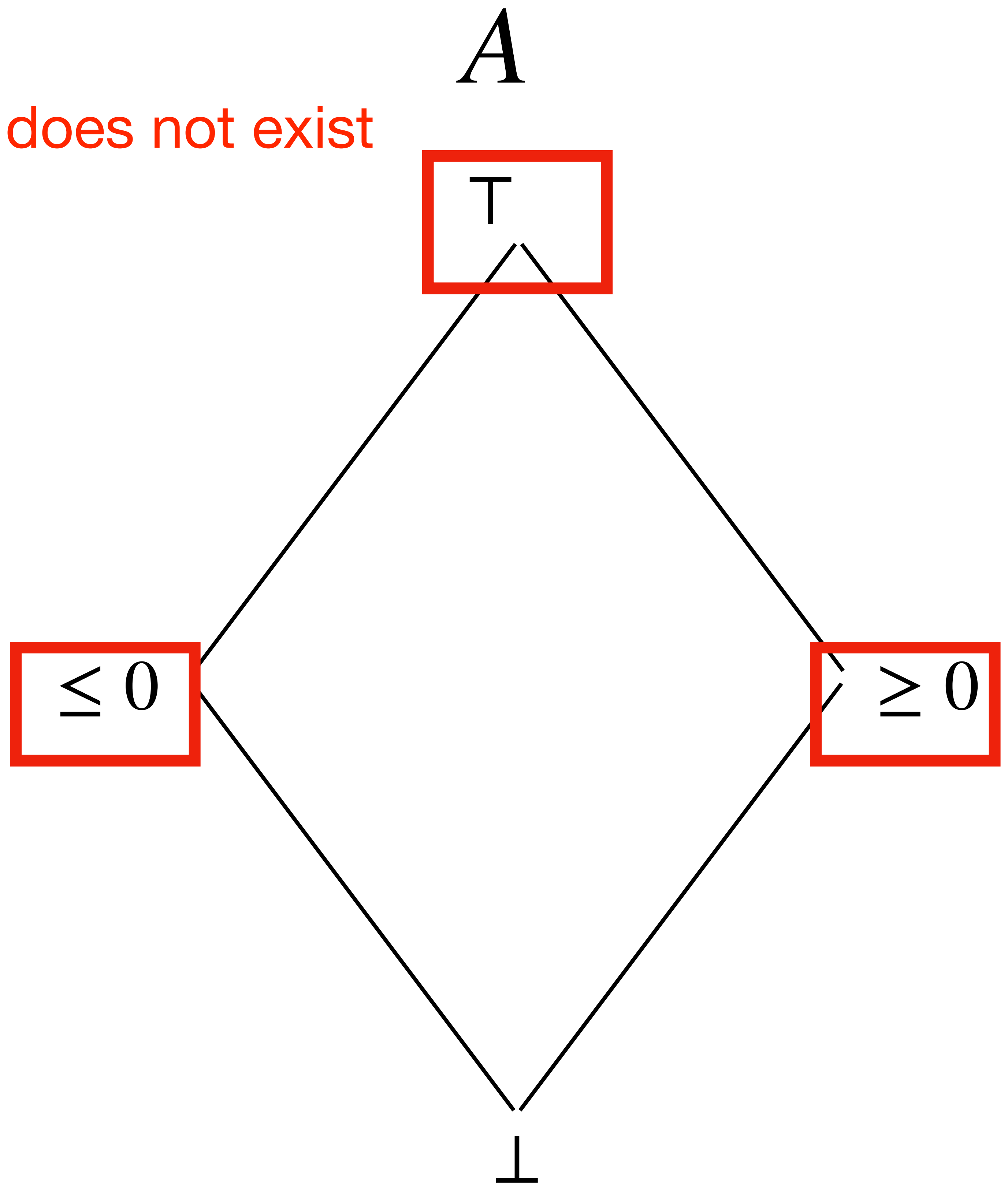
Abstraction function

Remark

To design an abstraction function $\alpha : C \to A$ the abstract domain must be closed under meet.

In this abstract domain the most precise element that approximates $\{0\}$ does not exist

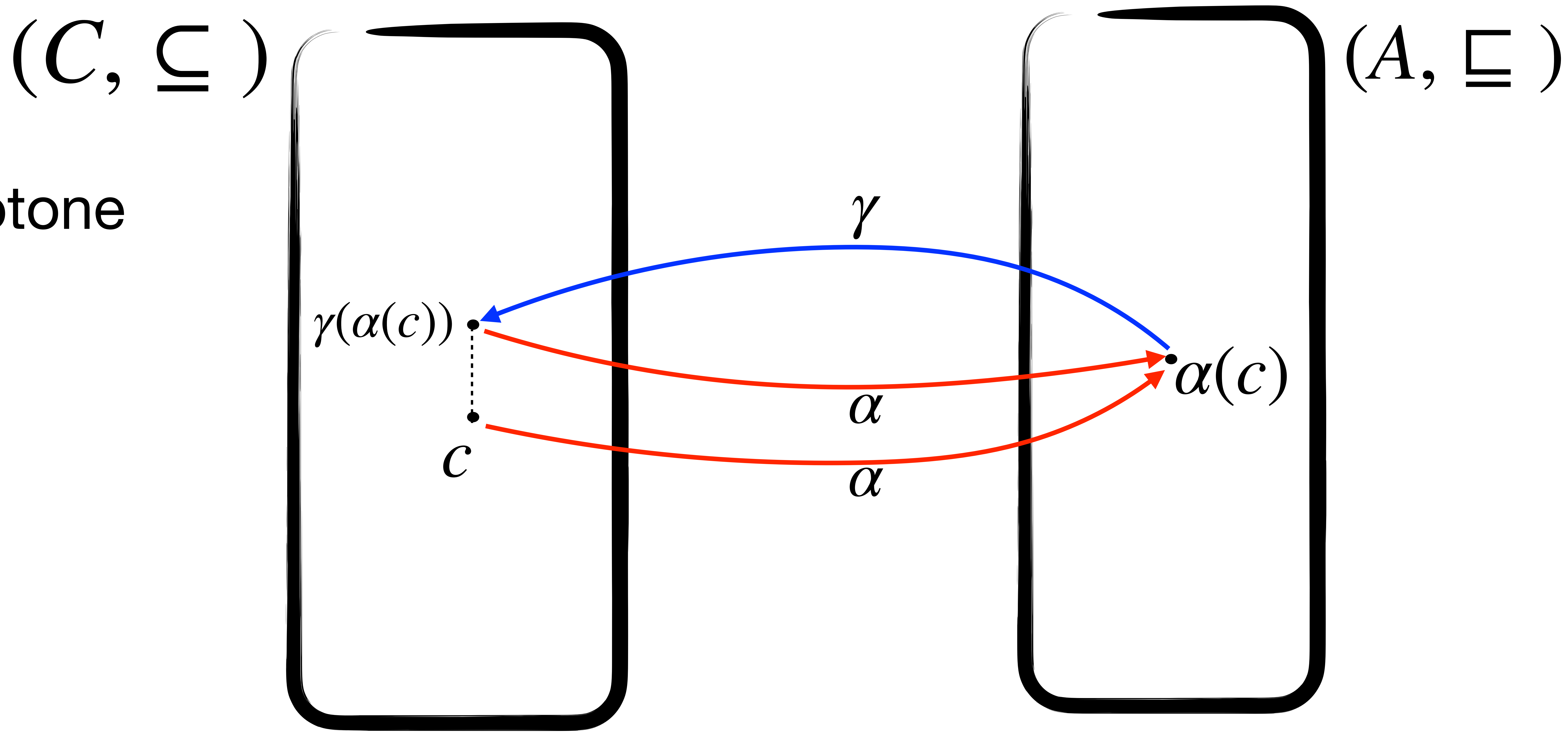




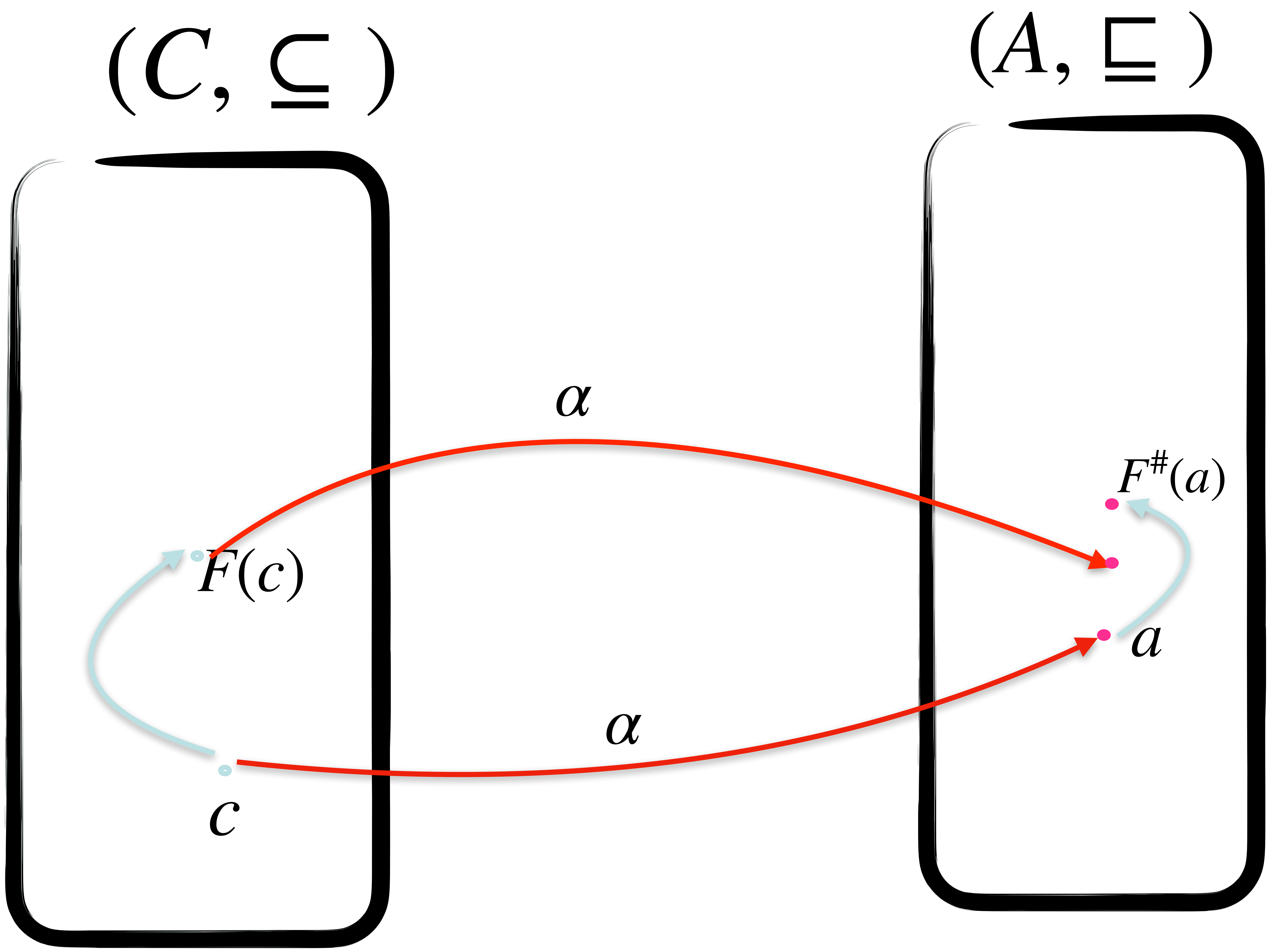
Abstract Interpretation
(AI)

Properties of Galois insertions

- α and γ are monotone
- $c \subseteq \gamma(\alpha(c))$
- $\alpha(\gamma(a)) = a$

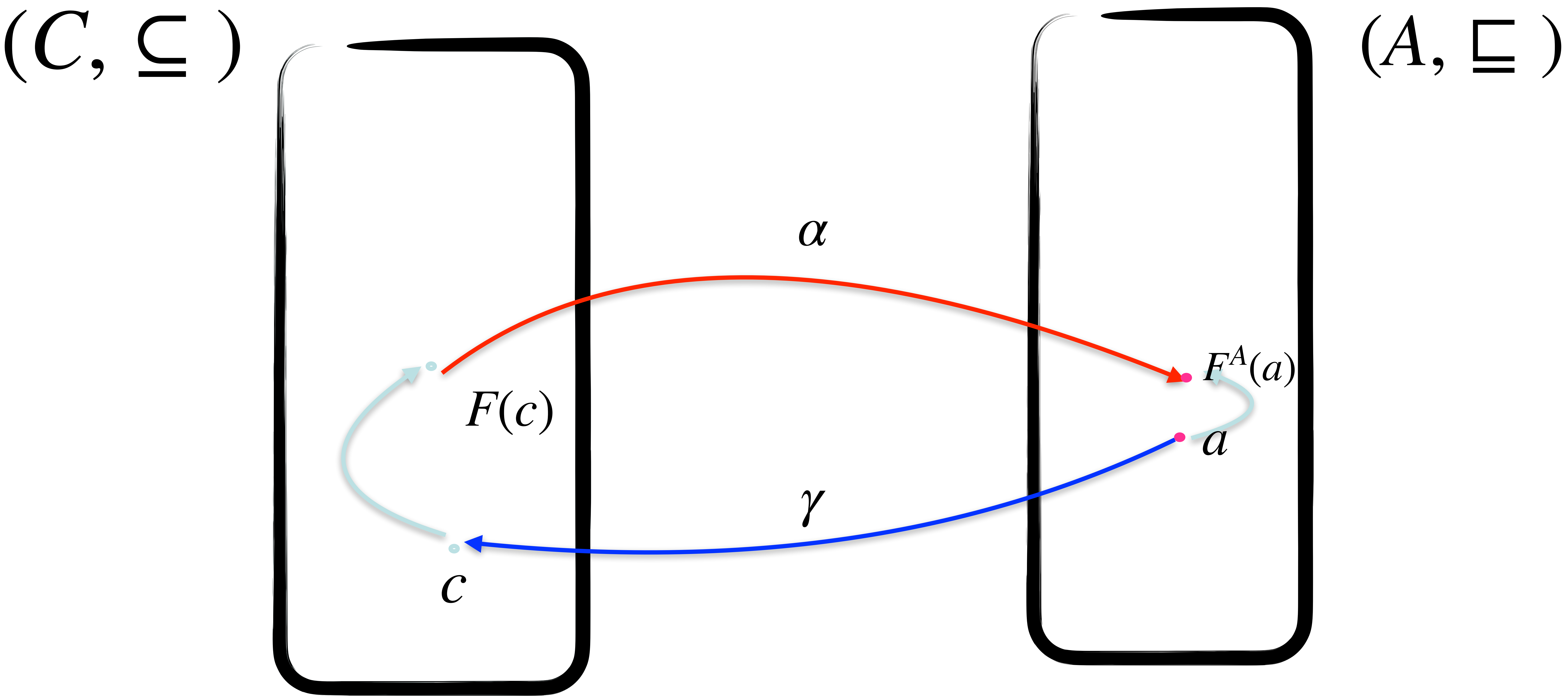


Correct approximations



$F^{\#}\alpha \sqsupseteq \alpha F$

Best correct approximation (bca)



$F^A \triangleq \alpha F \gamma$

Abstract operations: +

$$(\wp(\mathbb{Z}), \subseteq)$$

$$+ : \wp(Z) \to \wp(Z)$$

$$\{3,5\} + \{-2,4\} = \{1,7,3,9\}$$

$\{1,2,\ldots\}$ $\quad\{\ldots,-2,-1\}$ $\quad\top$

$> 0 \quad < 0 \quad > 0$

$$\{3\} + \{-2\} = \{1\}$$

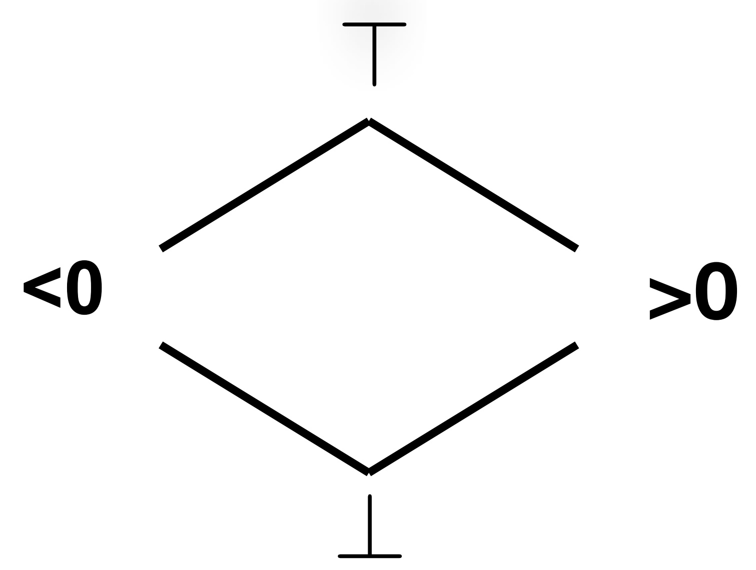
We lost precision

$\{1,2,\ldots\}$ $\quad\{1,2,\ldots\}$ $\quad\{2,3,\ldots\}$

$> 0 \quad > 0 \quad > 0$

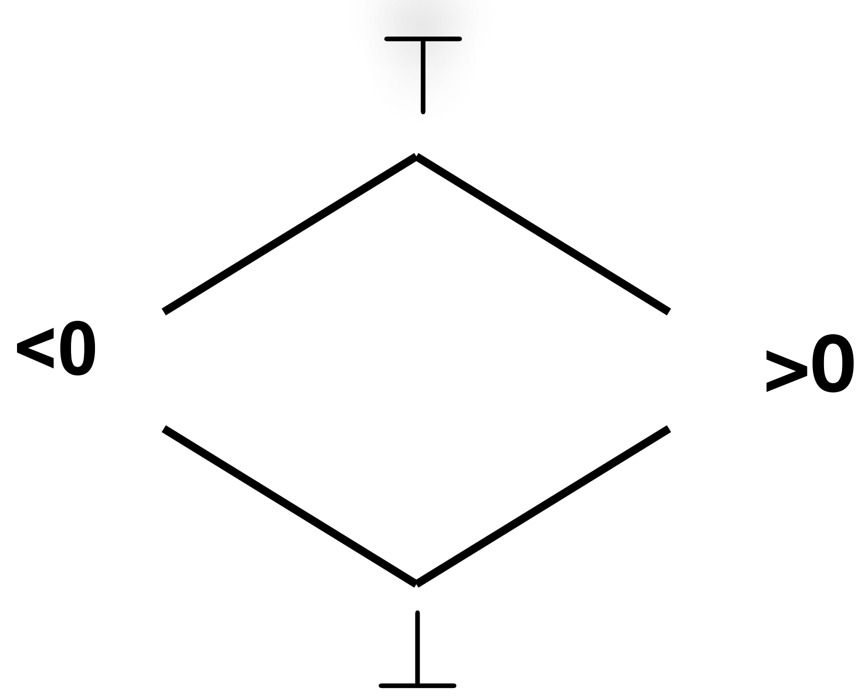
$$\{3\} + \{2\} = \{5\}$$

Precise result!



$+^\#$	⊥	<0	>0	T
⊥	⊥	⊥	⊥	⊥
<0	⊥	<0	T	T
>0	⊥	T	>0	T
T	⊥	T	T	T

Abstract operations: $\times$



$$(\wp(\mathbb{Z}), \subseteq)$$

$$\times : \wp(Z) \to \wp(Z)$$

$$\{3,5\} \times \{-2,4\} = \{-6, 12, -10, 20\}$$

> 0 $\quad < 0 \quad < 0$ ✓

$\{1,2,\ldots\}$ | $\{\ldots,-2,-1\}$ | $\{\ldots,-2,-1\}$

$$\{3\} \times \{-2\} = \{-6\}$$

Precise result!

> 0 $\quad > 0 \quad > 0$ ✓

$\{1,2,\ldots\}$ | $\{1,2,\ldots\}$ | $\{1,2,\ldots\}$

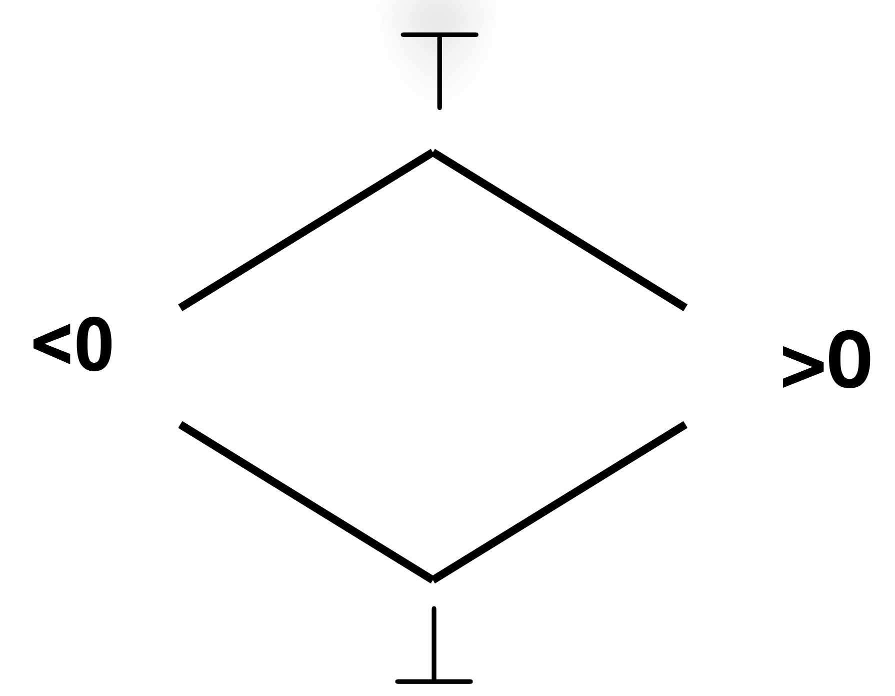
$$\{3\} \times \{2\} = \{6\}$$

Precise result!

$\times^{\#}$	⊥	<0	>0	⊤
⊥	⊥	⊥	⊥	⊥
<0	⊥	>0	<0	⊤
>0	⊥	<0	>0	⊤
⊤	⊥	⊤	⊤	⊤

Correctness

The abstract operations $+^{\#}$ and $\times^{\#}$ are correct on the domain Sign:



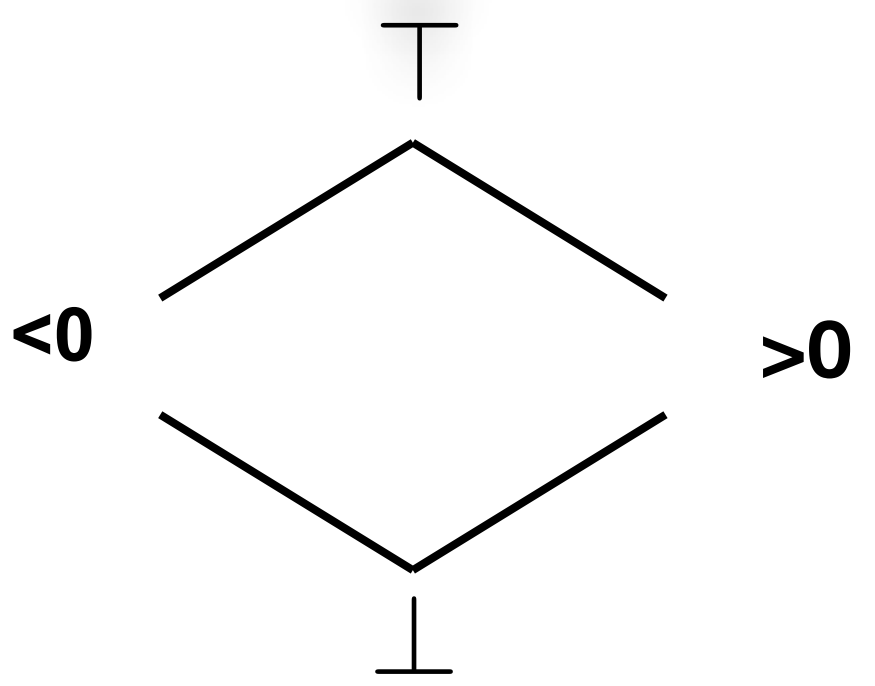
$$\forall P, Q \in \wp(\mathbb{Z}) \,.\, \alpha(P) +^{\#} \alpha(Q) \sqsupseteq \alpha(P + Q)$$

Remember $F^{\#}$ is correct on an abstract domain A
whenever it returns an approximation of the result of the concrete computation:

$$F^{\#}\alpha \sqsupseteq \alpha F$$

Completeness

The abstract operation $\times^{\#}$ has a very nice property on the domain Sign:



$$\forall P, Q \in \wp(\mathbb{Z}) \,.\, \alpha(P) \times^{\#} \alpha(Q) \overset{\not\sqsupseteq}{=} \alpha(P \times Q)$$

$F^{\#}$ is complete on an abstract domain A whenever it also holds:

$$F^{\#}\alpha = \alpha F$$

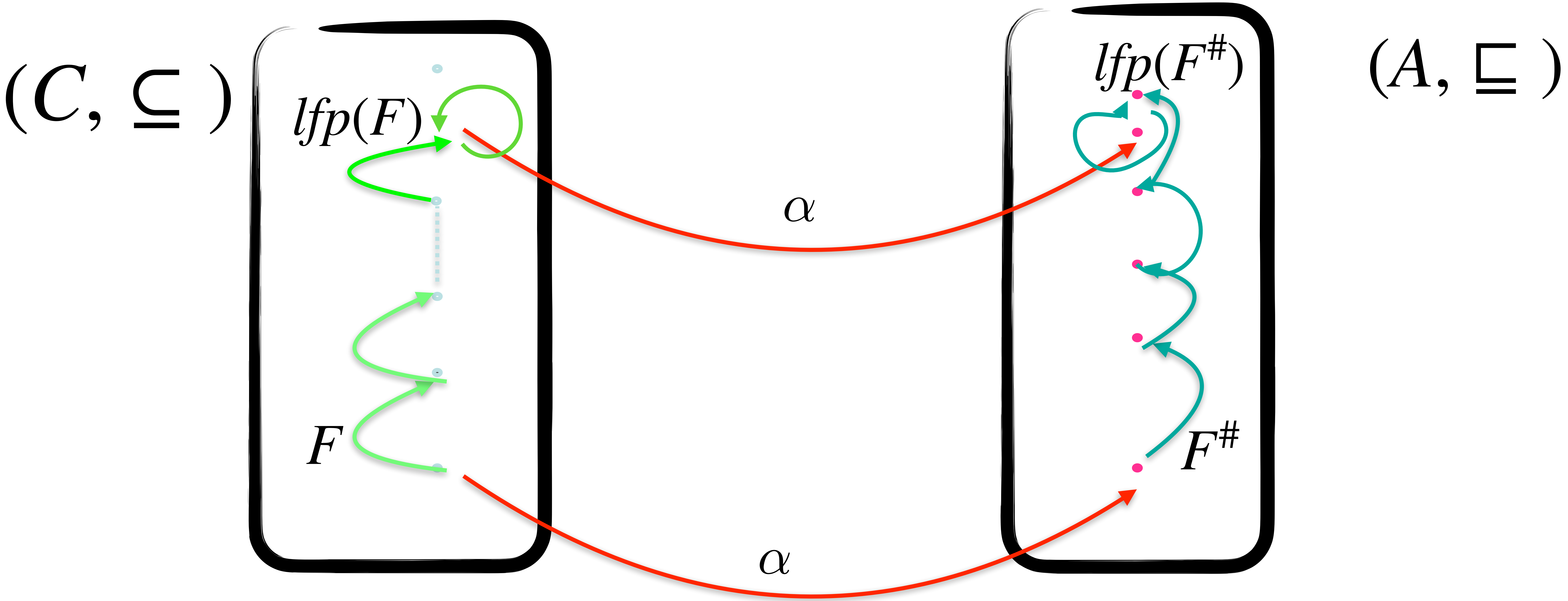
Completeness and bcas

$F^{\#}$ is complete $\implies$ $F^{\#} = F^{A}$

$$\alpha F = F^{\#}\alpha \implies F^{A} = \alpha F\gamma = F^{\#}\alpha\gamma = F^{\#}$$

Fixpoint computation approximation

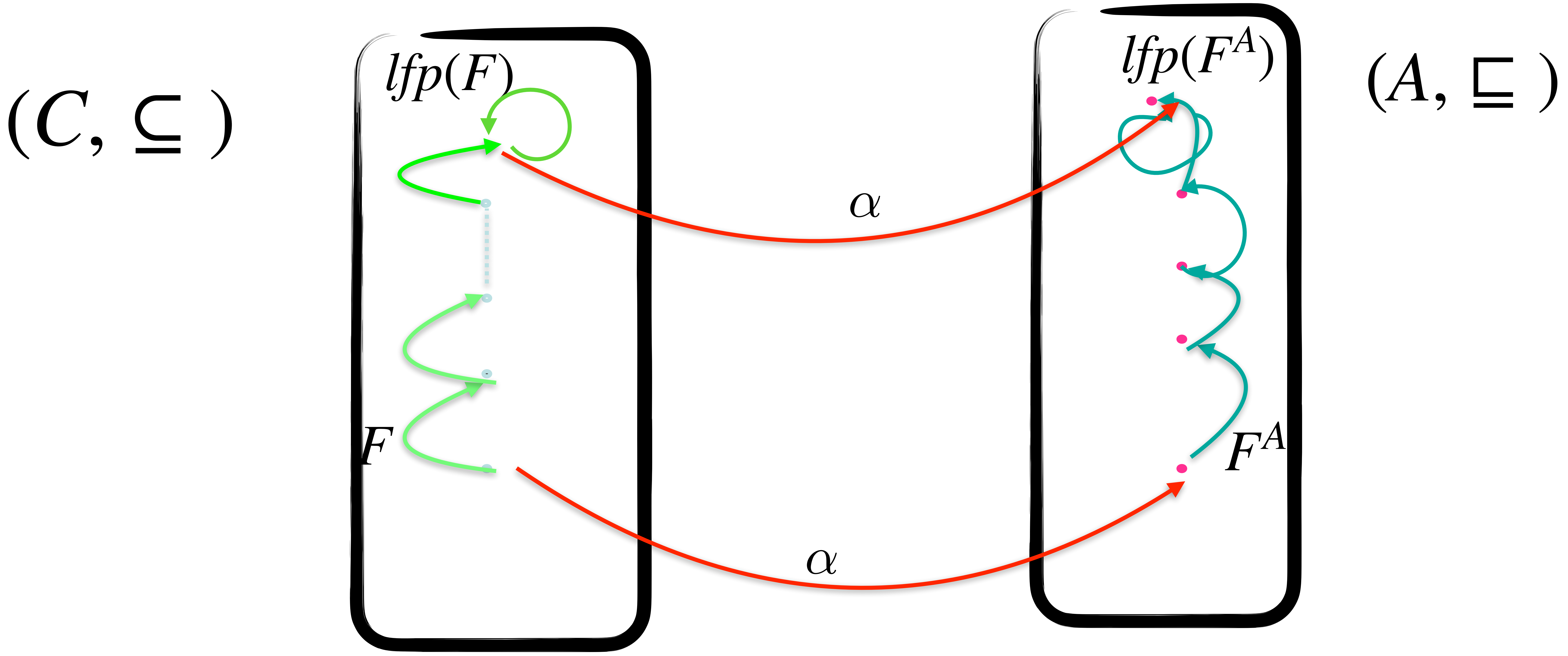
If F monotone and $F^{\#}$ correct



$lfp(F^{\#})$ is a correct over approximation of $lfp(F)$

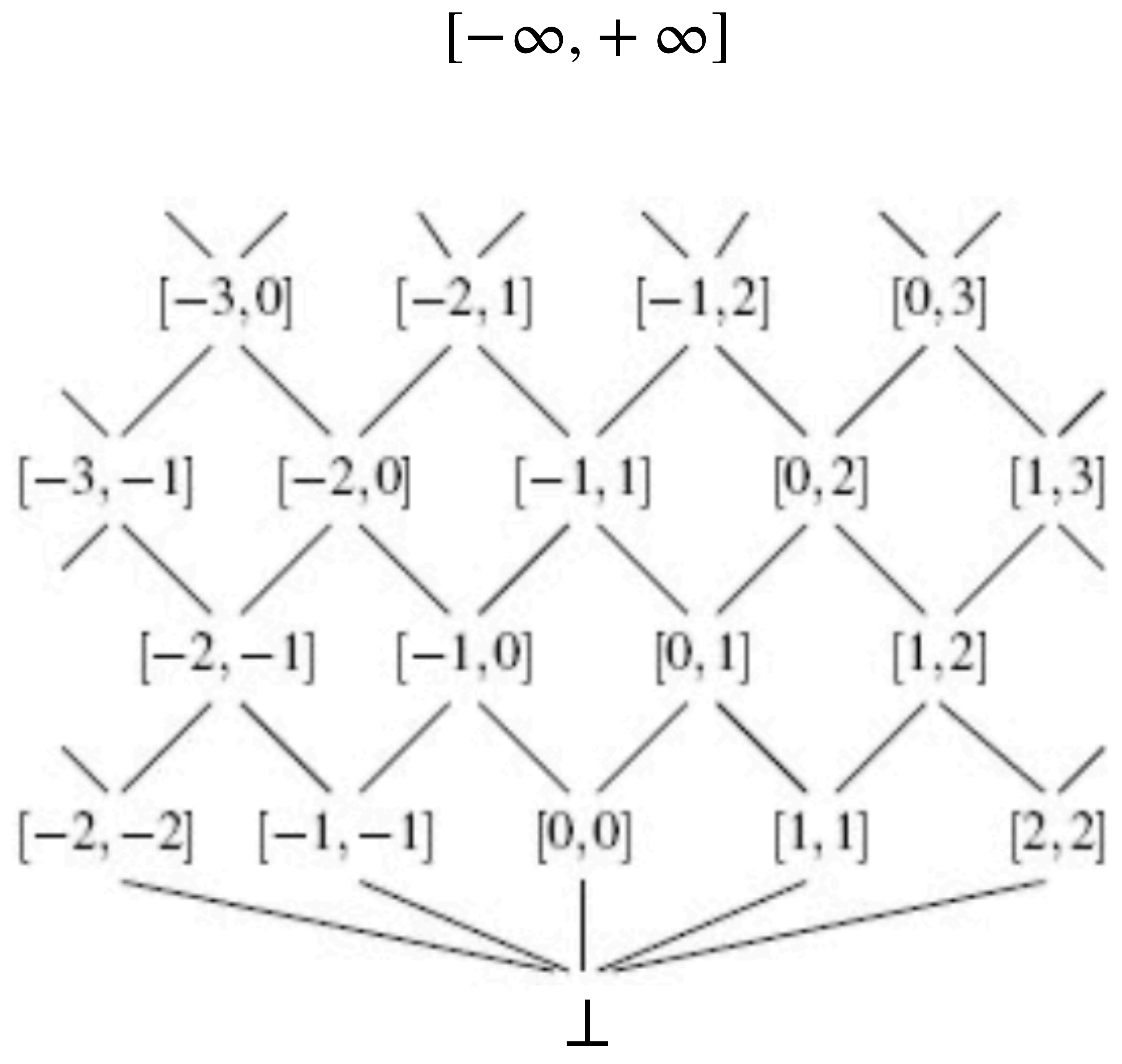
Fixpoint computation approximation

If F monotone and F^A is complete



Abstract domains

Intervals



Elements of A:

- $\bot$ the empty set of values
- $[n_0, n_1],\ n_0 \in (\mathbb{Z} \cup \{-\infty\}),\ \ n_1 \in (\mathbb{Z} \cup \{+\infty\}), n_0 \leq n_1$

$\sqsubseteq$ is the interval inclusion

$$\begin{aligned}
\gamma(\bot) &= \{\} \\
\gamma([n_0, n_1]) &= \{\ n \in \mathbb{Z}\ \mid n_0 \leq n \leq n_1\} \\
\gamma([-\infty, n_1]) &= \{\ n \in \mathbb{Z} \mid n \leq n_1\} \\
\gamma([n_0, +\infty]) &= \{\ n \in \mathbb{Z} \mid n_0 \leq n\} \\
\gamma([-\infty, +\infty]) &= \mathbb{Z}
\end{aligned}$$

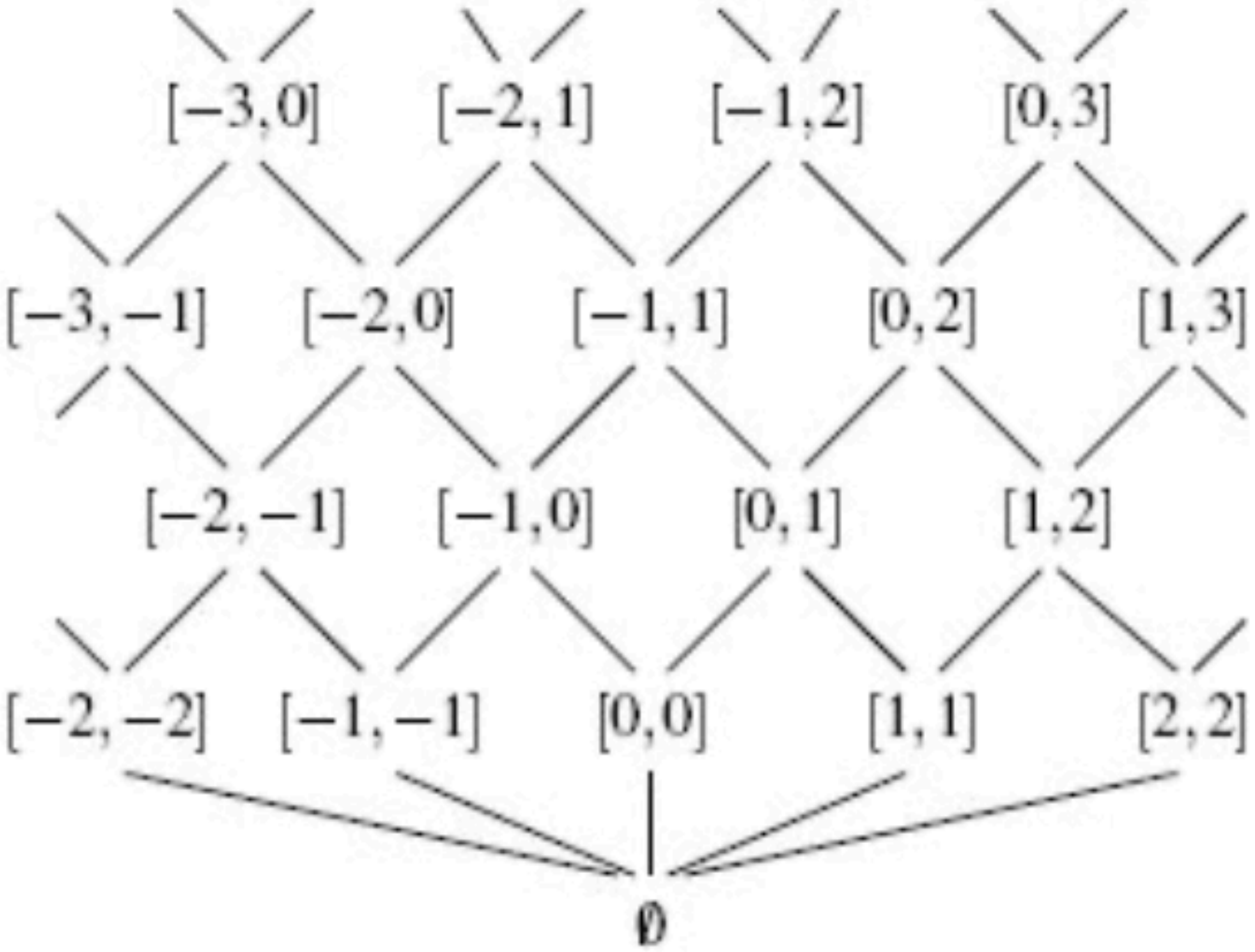
$$\begin{aligned}
&\alpha(c) = \bot \text{ if } c = \emptyset, \\
&\alpha(c) = [min(c), max(c)] \text{ if } c \neq \emptyset, min(c) \text{ and } max(c) \text{ exists} \\
&\alpha(c) = [min(c), +\infty] \text{ if } c \neq \emptyset, min(c) \text{ exists} \\
&\alpha(c) = [-\infty, max(c)] \text{ if } c \neq \emptyset, max(c) \text{ exists} \\
&\alpha(c) = [-\infty, +\infty] \text{ otherwise}
\end{aligned}$$

$+^A$ and $\times^A$ are complete on Int

$$[n, m] +^A [p, r] = [n + p, m + r]$$

$$[n, m] \times^A [p, r] = [n \times p, m \times r]$$

if all positives, otherwise pay attention



$[1,6]$ $[-3,1]$ $[-2,7]$

Precise result!

$$\{1,4,6\} + \{-3,1\} = \{-2,1,2,3,5,7\}$$

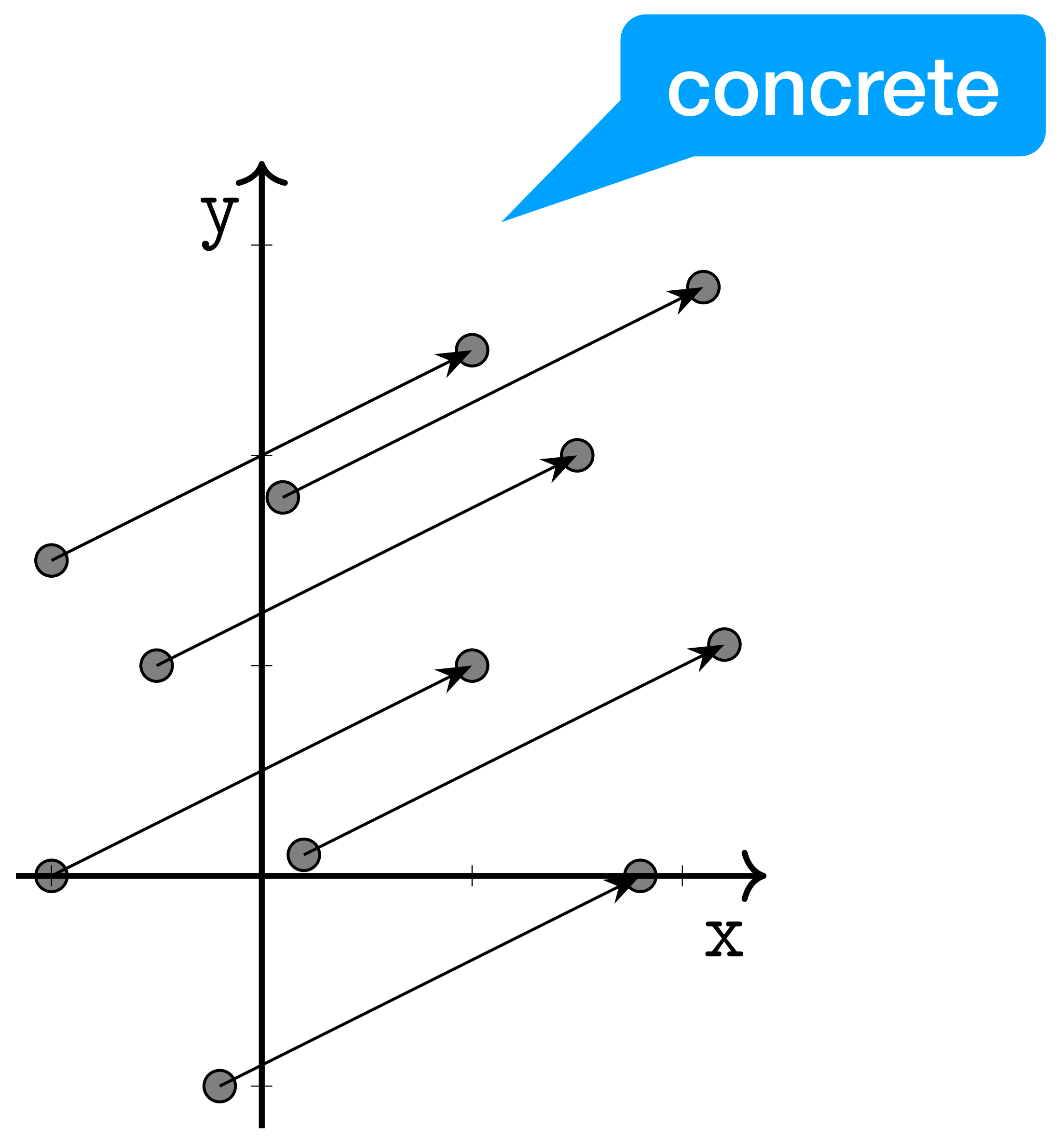
Tests are not complete on Int

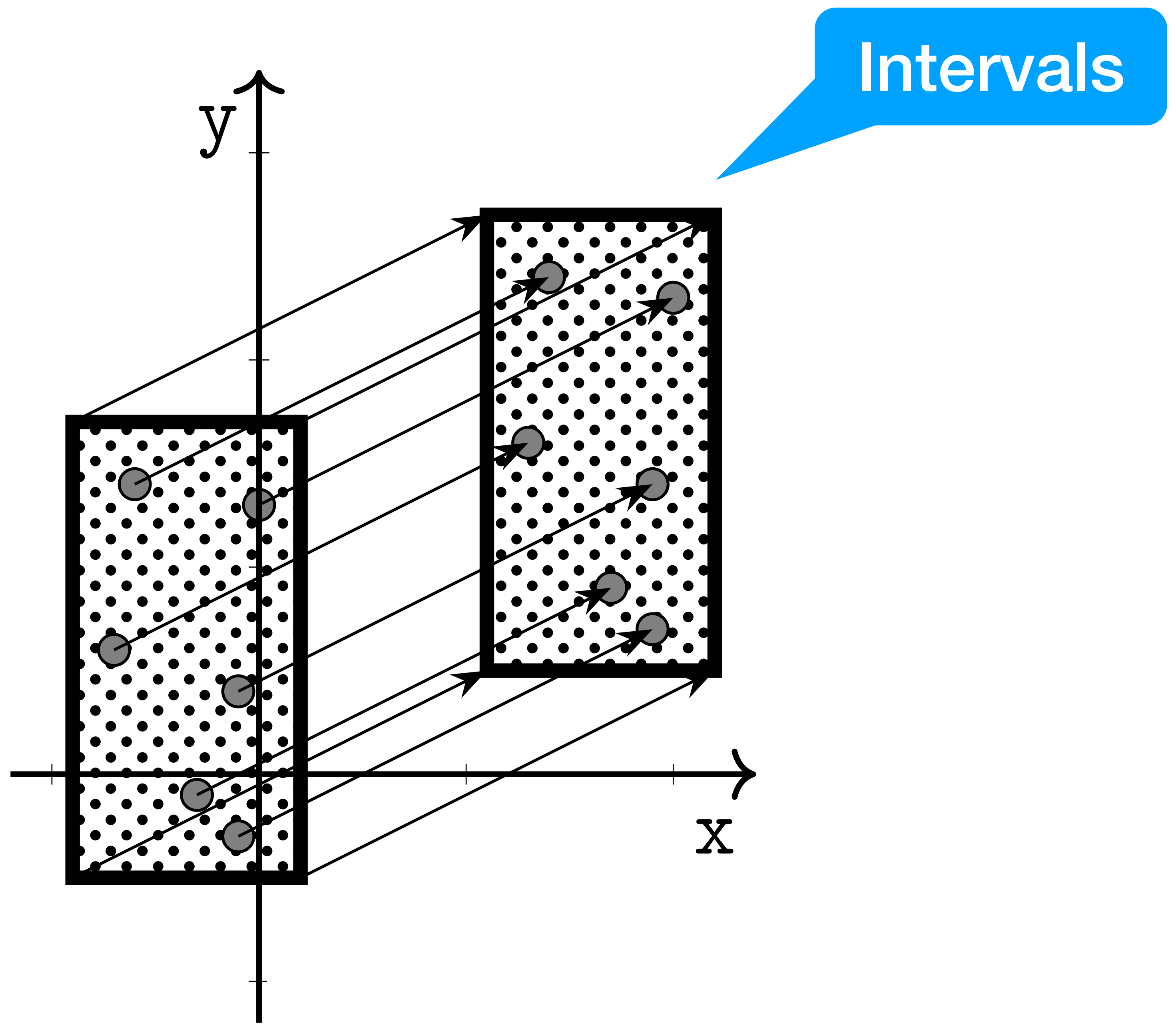
$[-7, -1]$ $\sqsupseteq$

Concrete $(x < 0)$

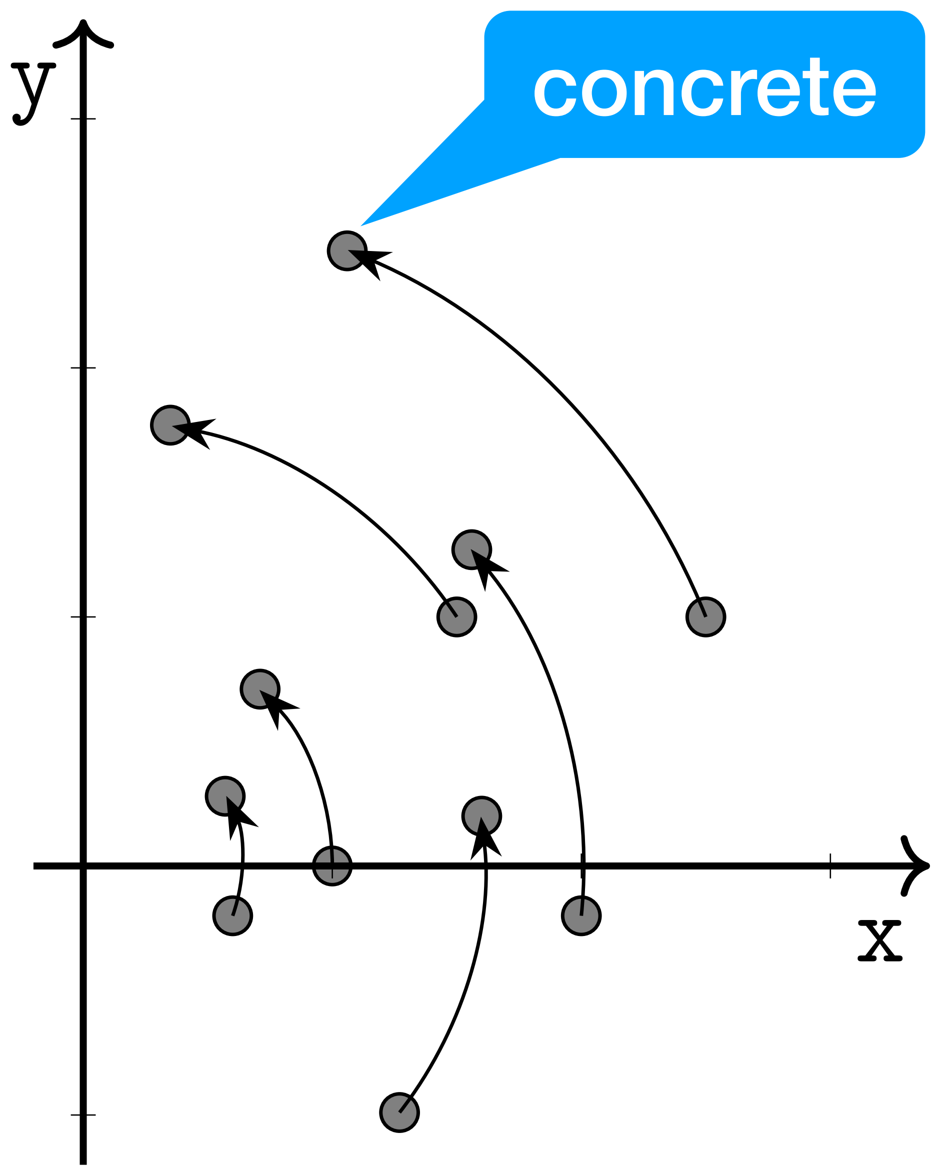
$P = \{-7, 0\}$ $A(P) = [-7, 0]$ $(x < 0)$ $\rightarrow$ $[-7, -7] = A([\![c]\!]P)$ $= [\![(x < 0)]\!]P$

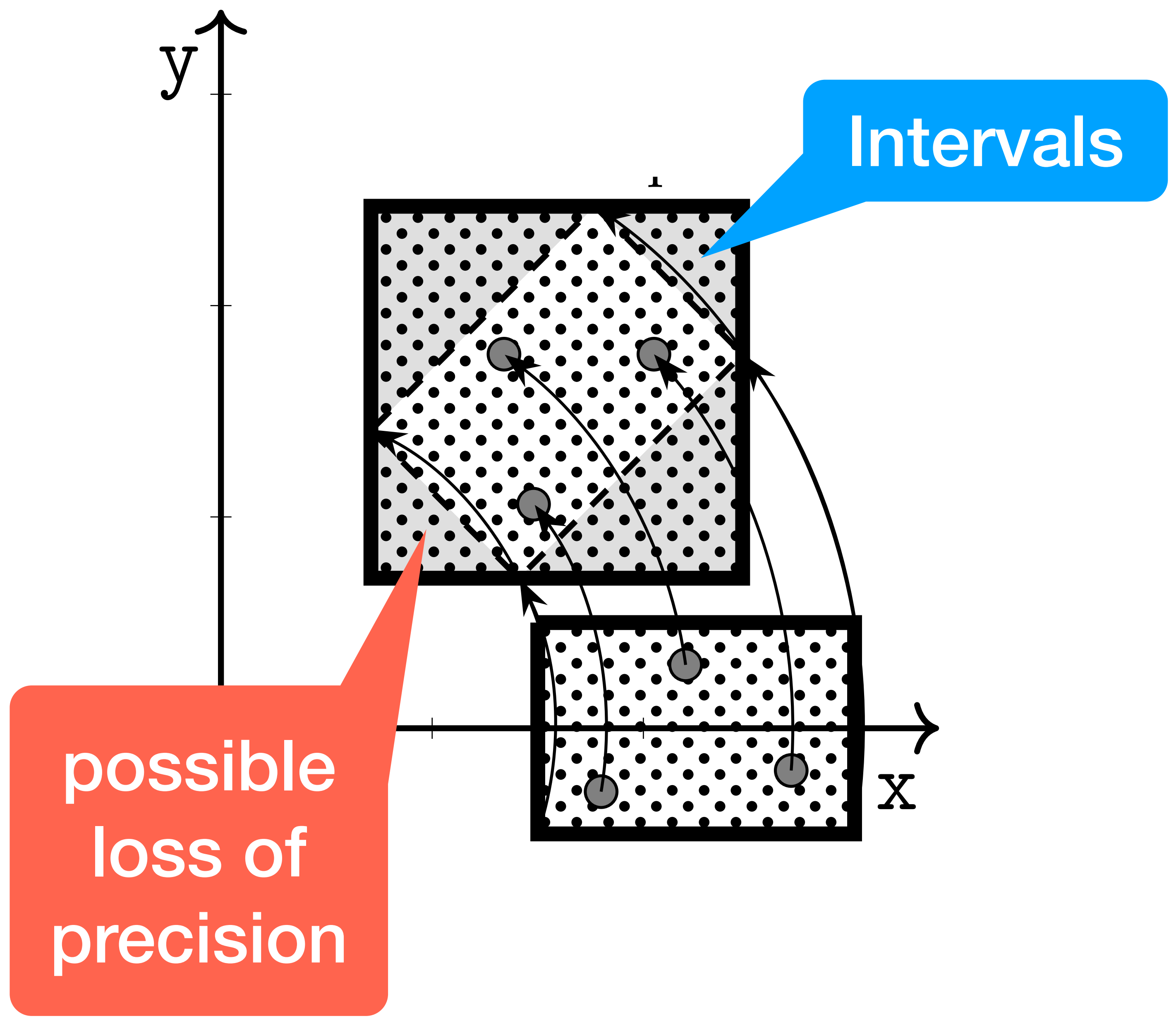
Example: translation





Example: rotation



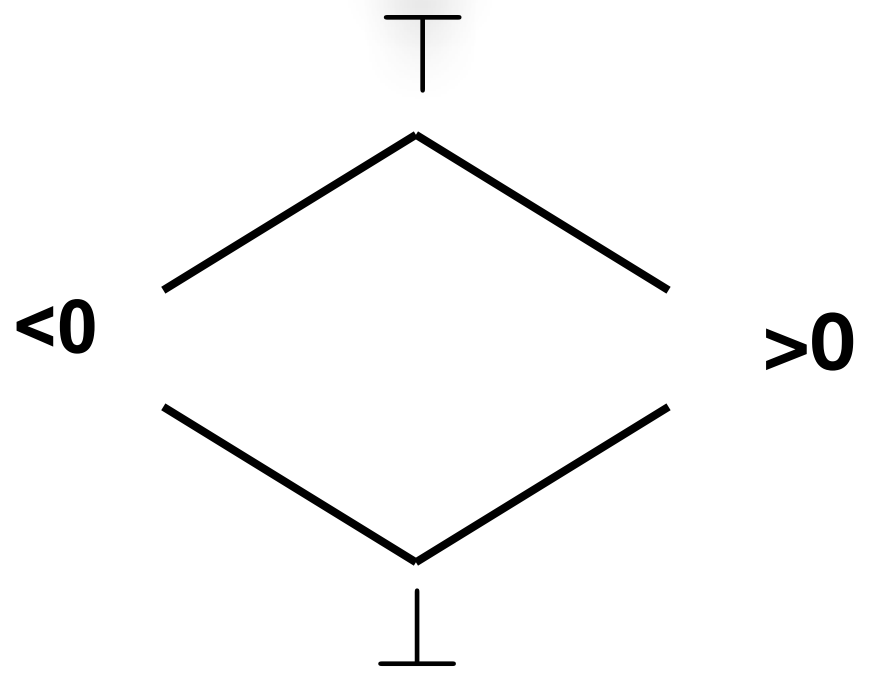


Composition of bcas

The composition of bca is not always a bca

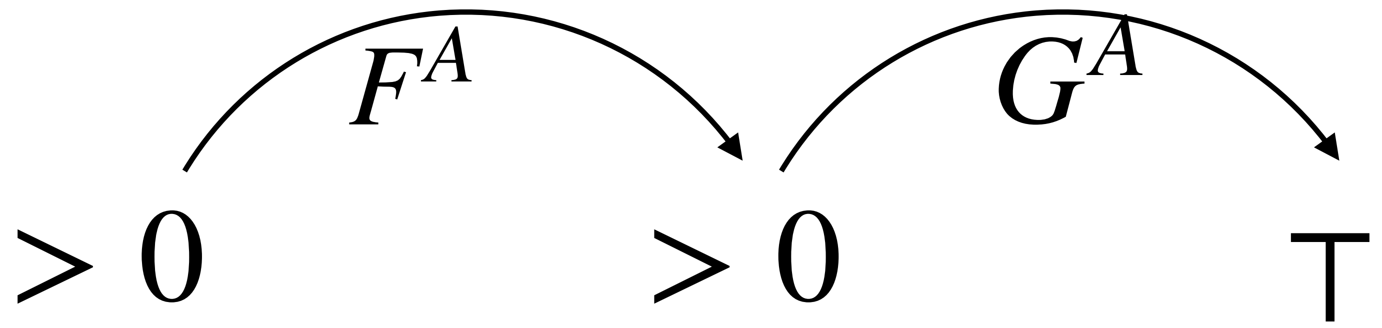
For F^A and G^A bca, in general

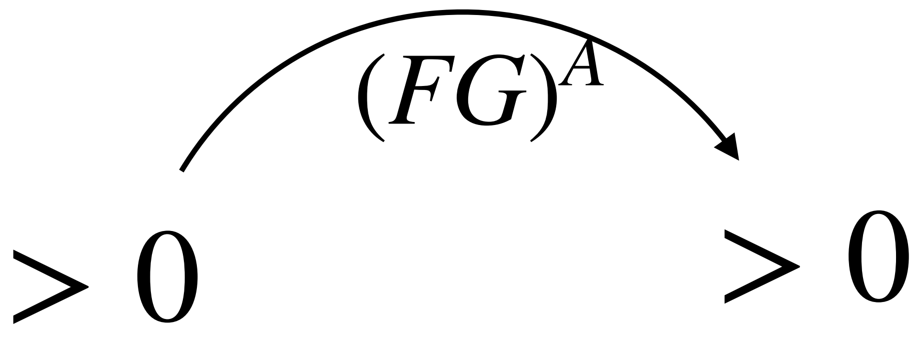
$$F^A G^A \neq (FG)^A \quad \text{Indeed} \quad \alpha F \gamma \alpha G \gamma \sqsupseteq \alpha F G \gamma \quad \text{because} \quad \gamma\alpha \sqsupseteq \text{id}$$



Example

$$F = \_ + 1 \quad G = \_ - 1 \quad FG = \text{id}$$





Composition of complete abstractions

The composition of complete abstractions is always complete

For $F^\#$ and $G^\#$ complete abstractions

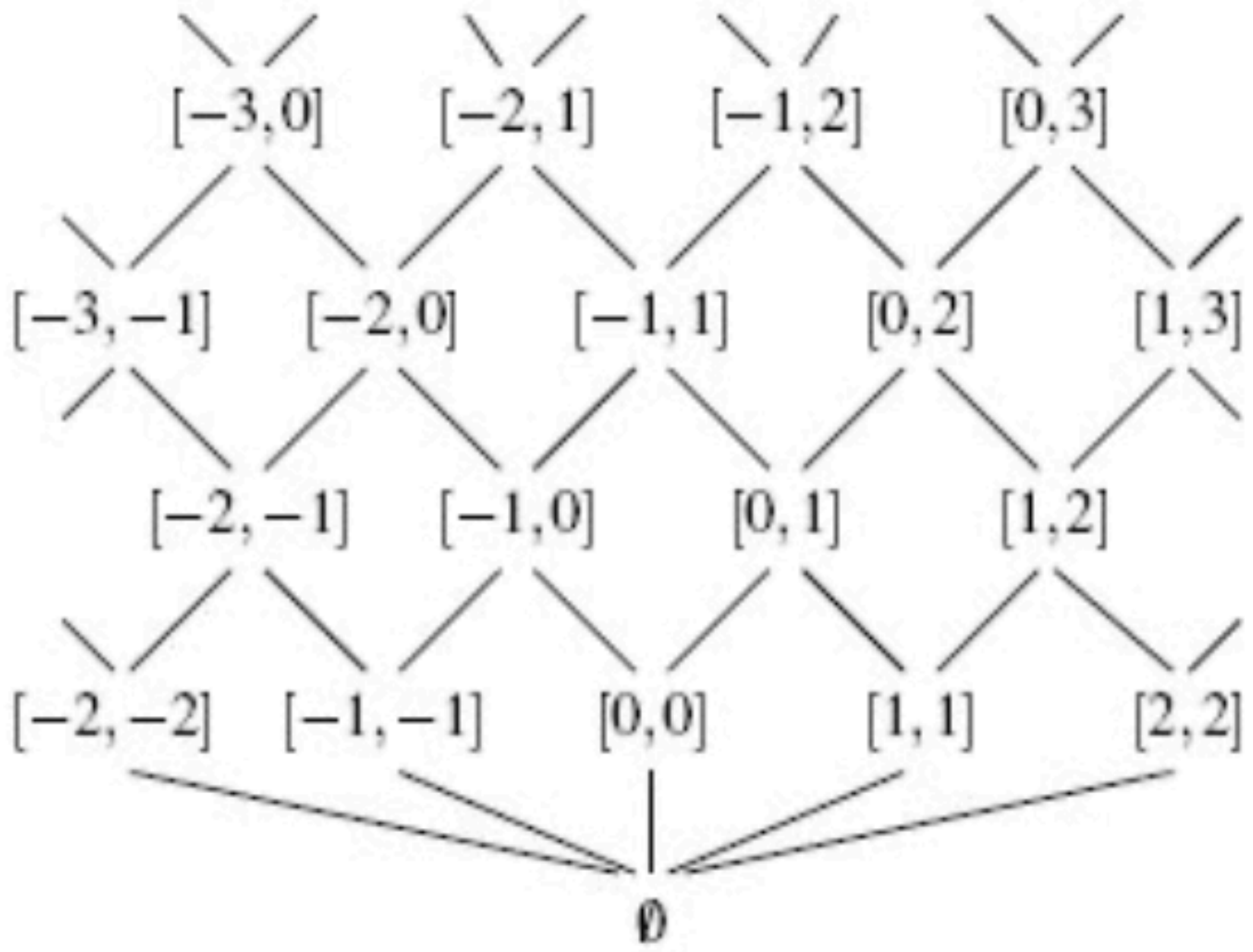
$$F^\# G^\# \alpha = F^\# \alpha G = \alpha F G$$

Example

$$F = \_ + 1 \quad G = \_ - 1 \quad FG = \text{id}$$

$[2,3] \xrightarrow{F^\#} [3,4] \xrightarrow{G^\#} [2,3]$

$[2,3] \xrightarrow{(FG)^A} [2,3]$



Non-relational domains

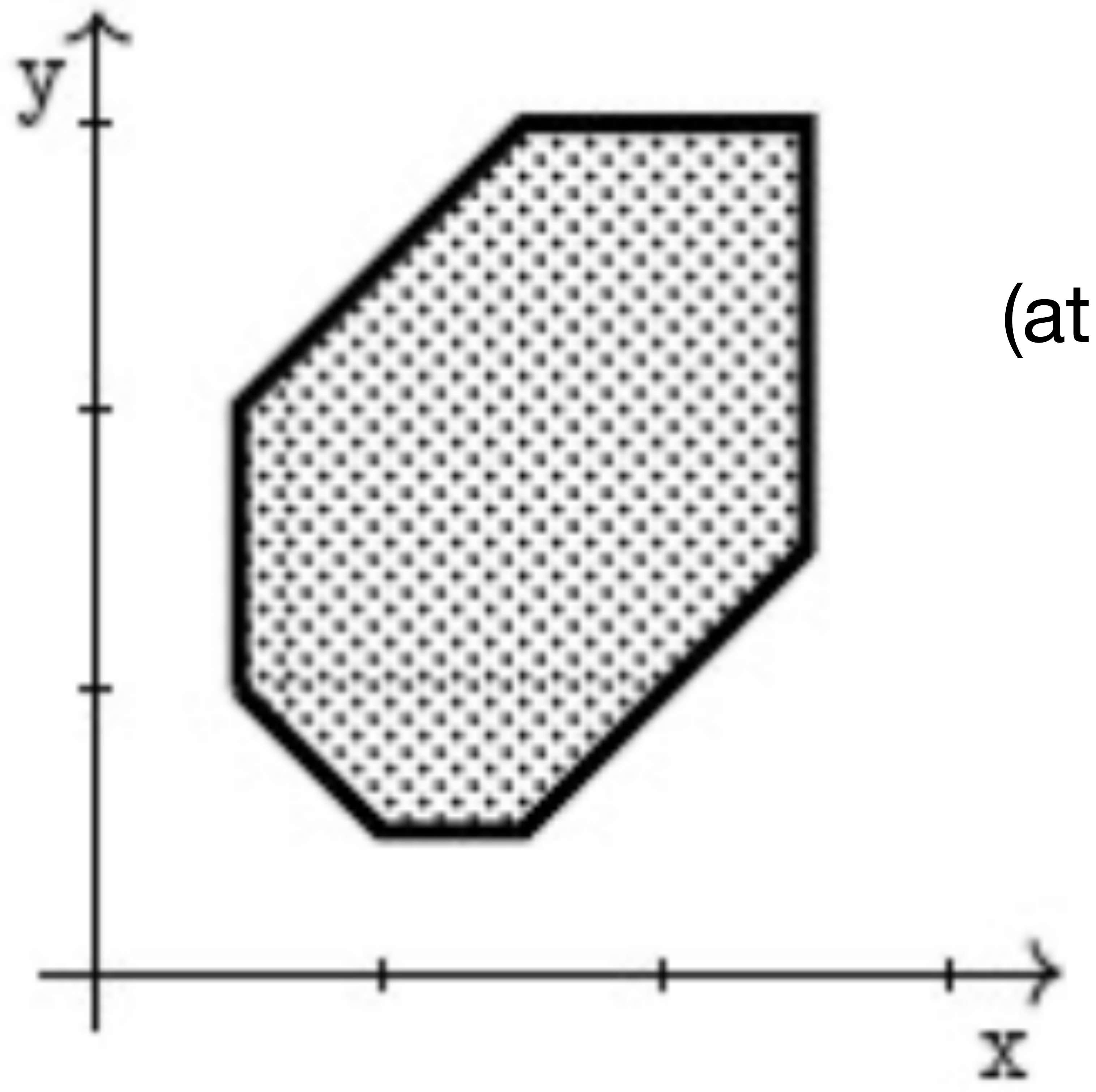
The domains of Sign and Interval are non-relational domains

They cannot track relations between variables values

The set of states

$$\{[x \mapsto 1, y \mapsto 6] \quad [x \mapsto 3, y \mapsto 8] \quad [x \mapsto 10, y \mapsto 15]\} \xrightarrow{\alpha} [x \mapsto [1,10], y \mapsto [6,15]] \xrightarrow{\gamma} \{[x \mapsto 1, y \mapsto 6] \quad [x \mapsto 1, y \mapsto 7] \quad \dots\}$$

Relational domain
Octagon domain



sets of numerical constraints of the form

$$\pm x \pm y \leq c$$

(at most two variables per constraint, with unit coefficients)

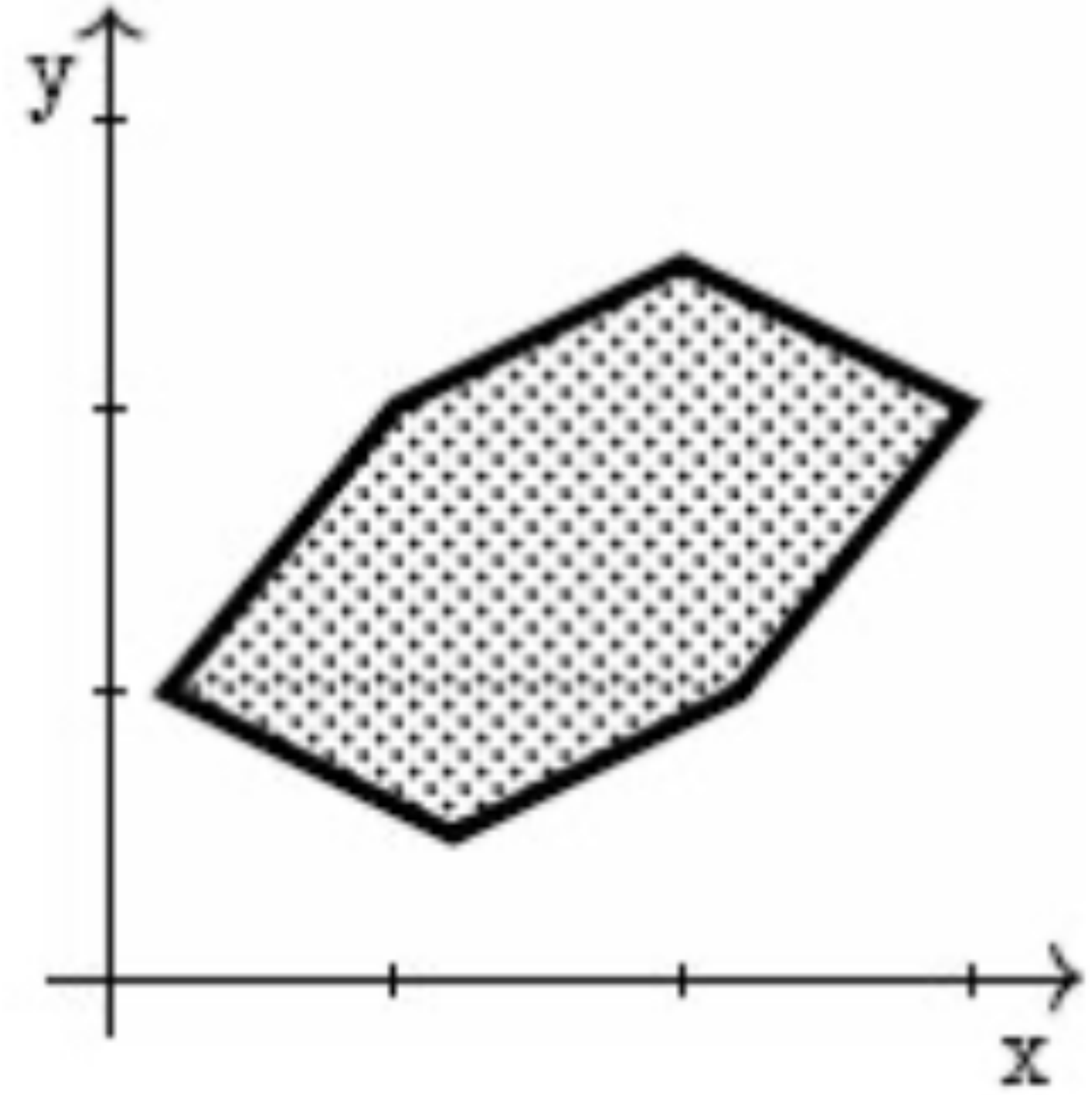
The set of states

$$\{[x \mapsto 1, y \mapsto 6]$$
$$[x \mapsto 3, y \mapsto 8]$$
$$[x \mapsto 10, y \mapsto 15]\}$$

α →

$$x \leq 10$$
$$x \geq 1$$
$$y \leq 15$$
$$y \geq 6$$
$$y - x = 5$$

Relational domain
Convex Polyhedra domain



sets of numerical constraints of the form

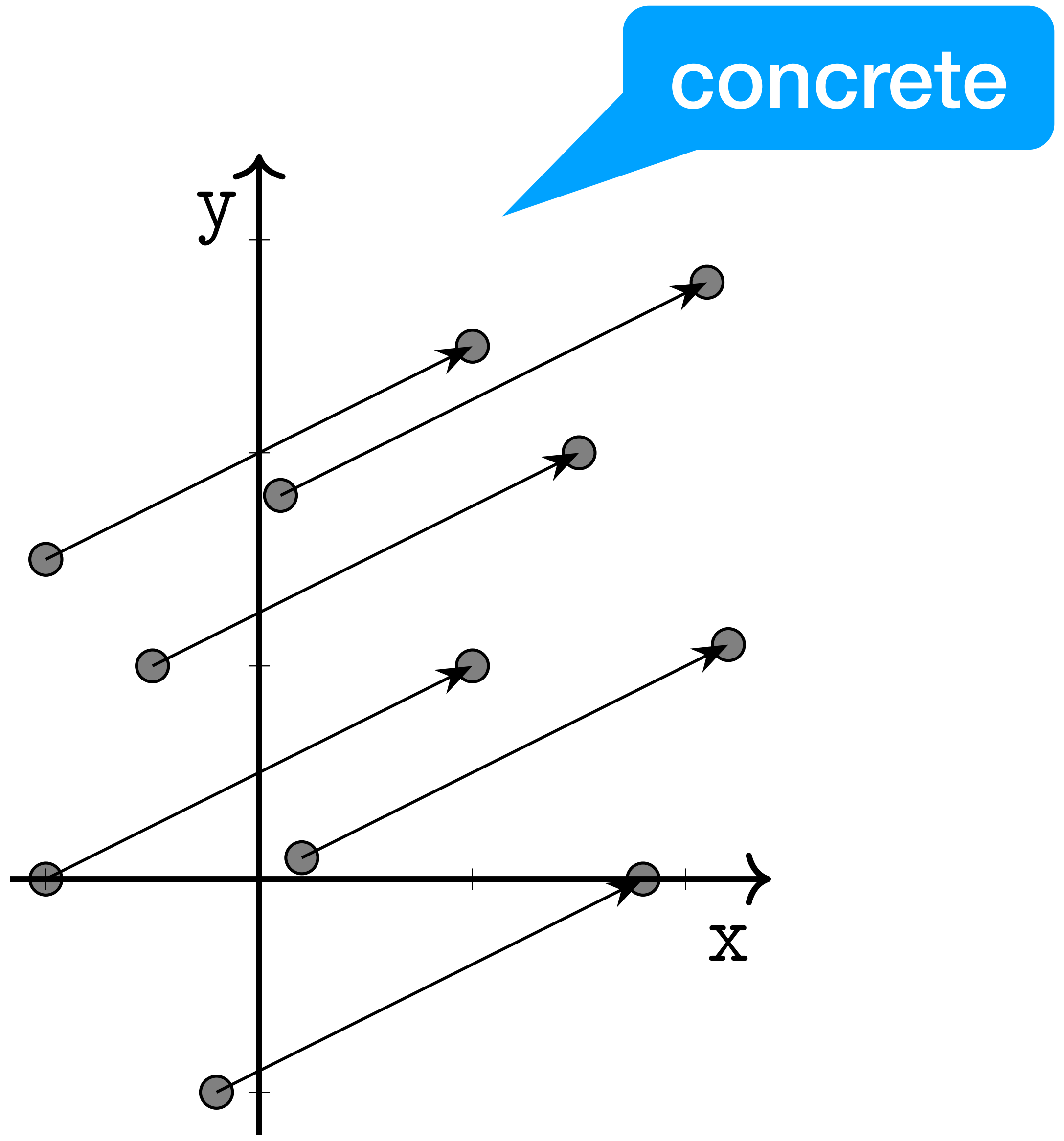
$$c_1 x + c_2 y \leq c$$

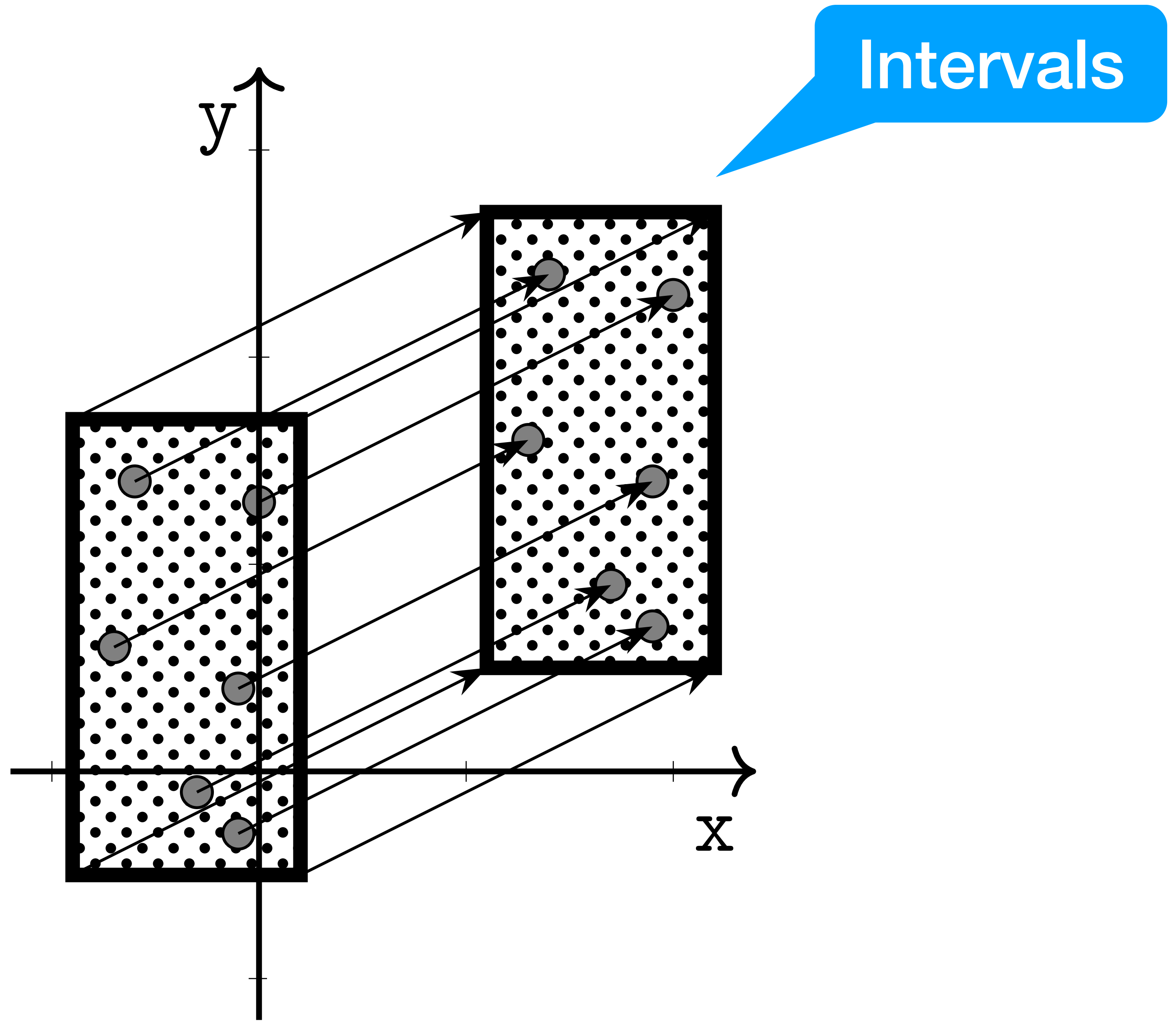
(at most two variables per constraint,
with unit coefficients)

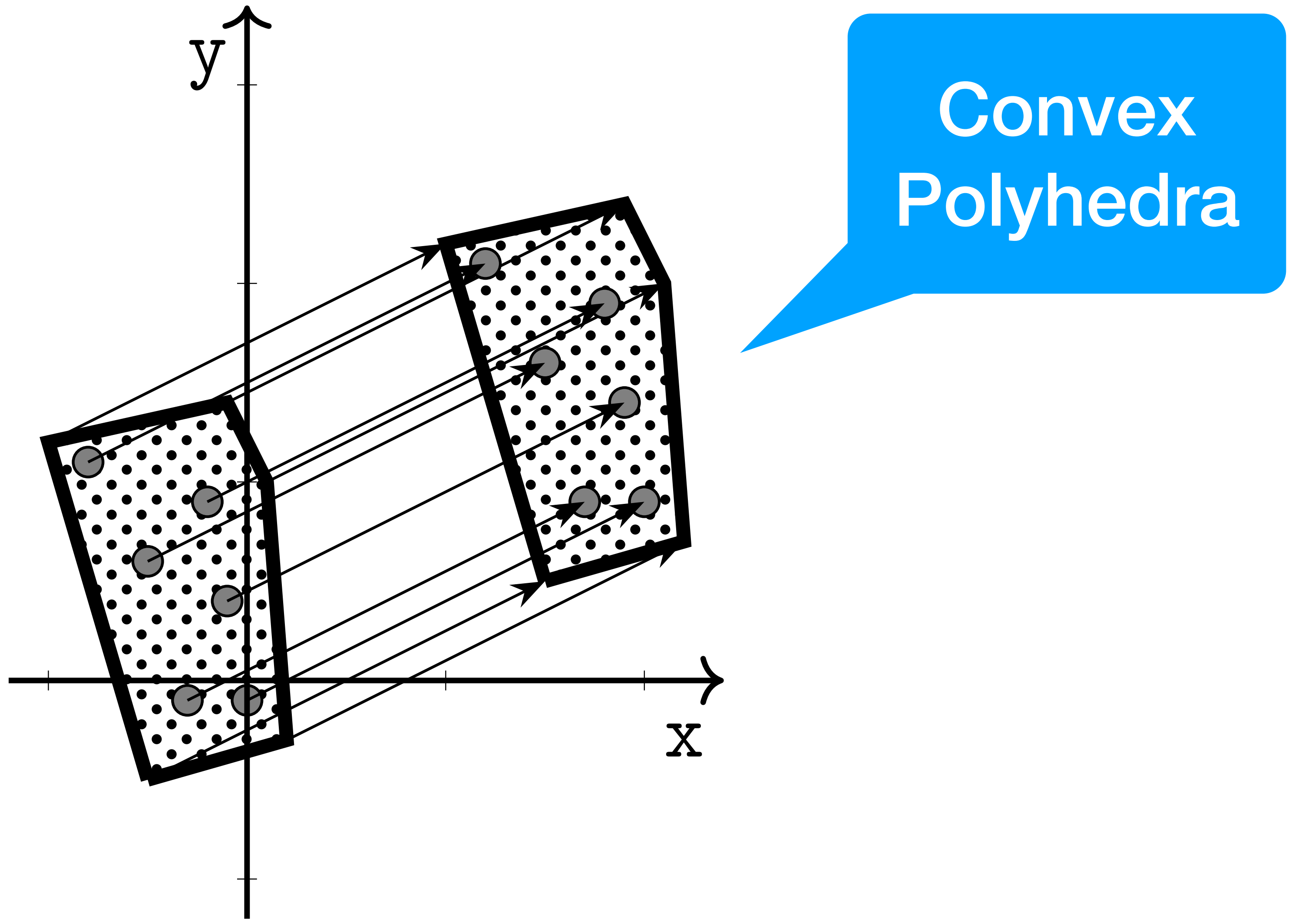
does not admit an abstraction map

best abstraction of ○ ?

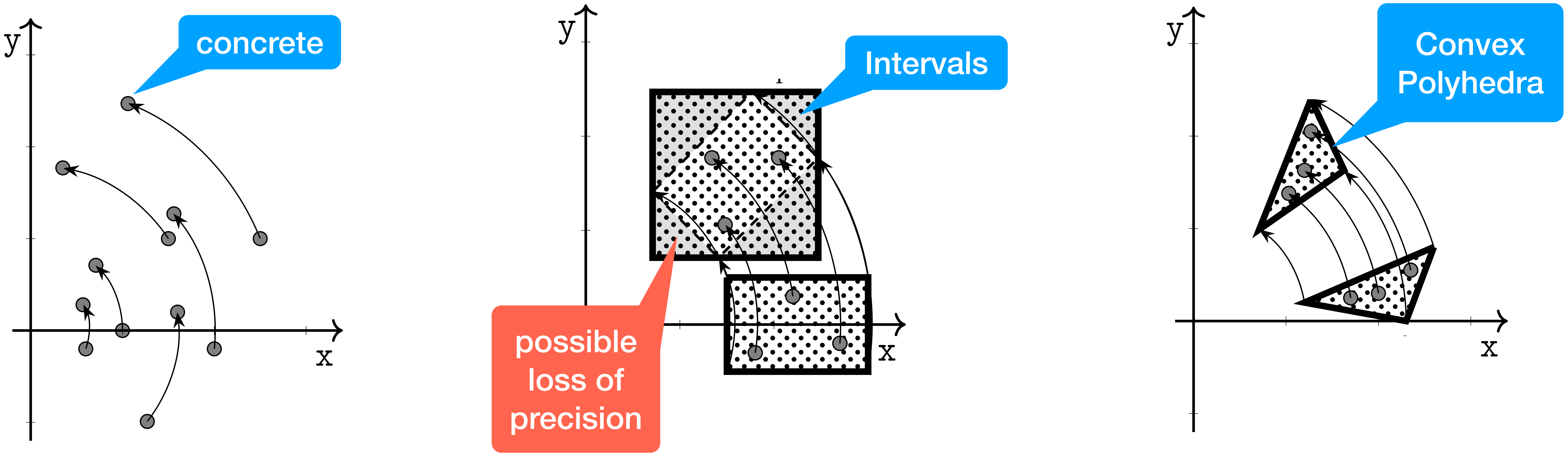
Example: translation







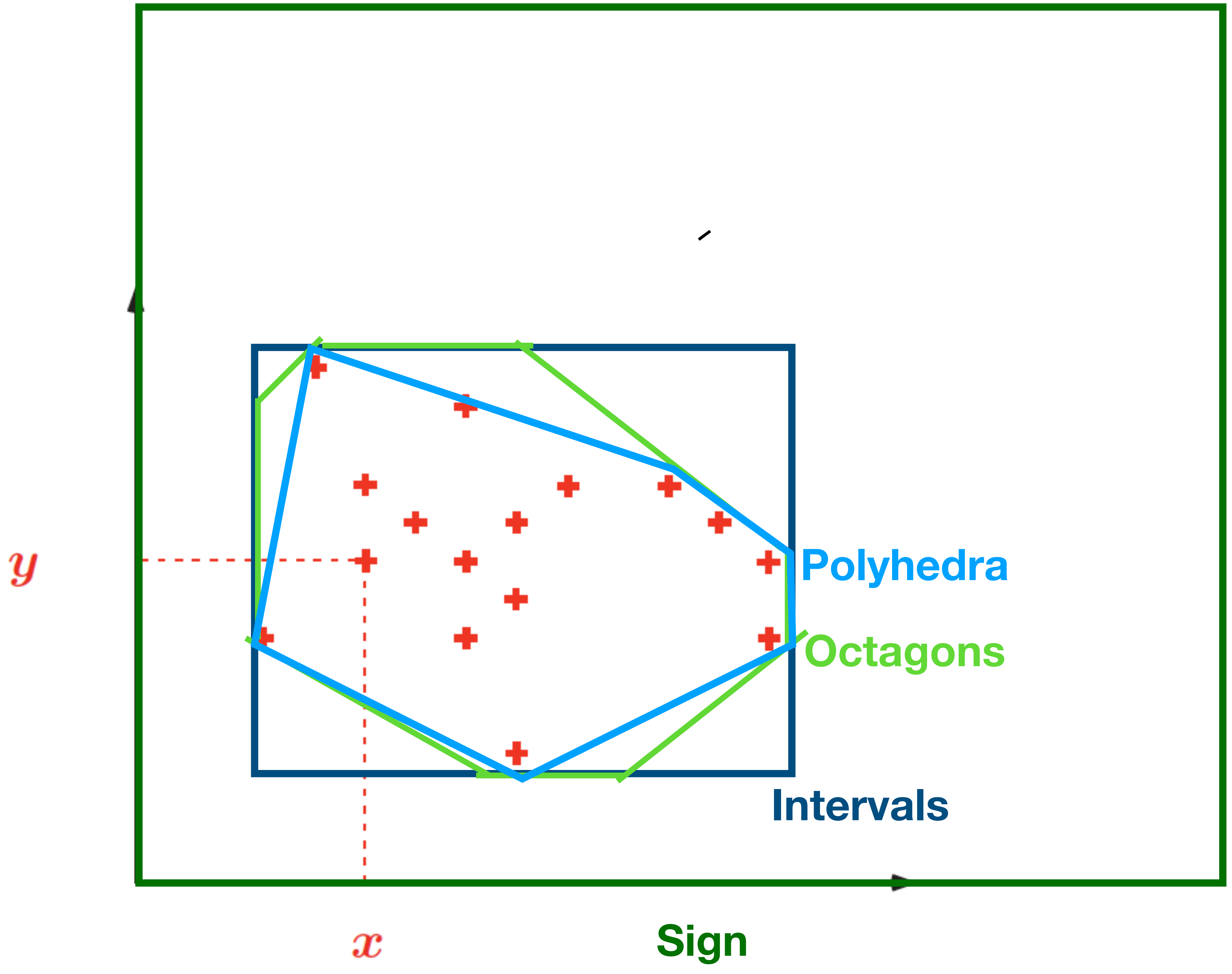
Example: rotation



Refinements of abstraction

An (in)-finite set of points :

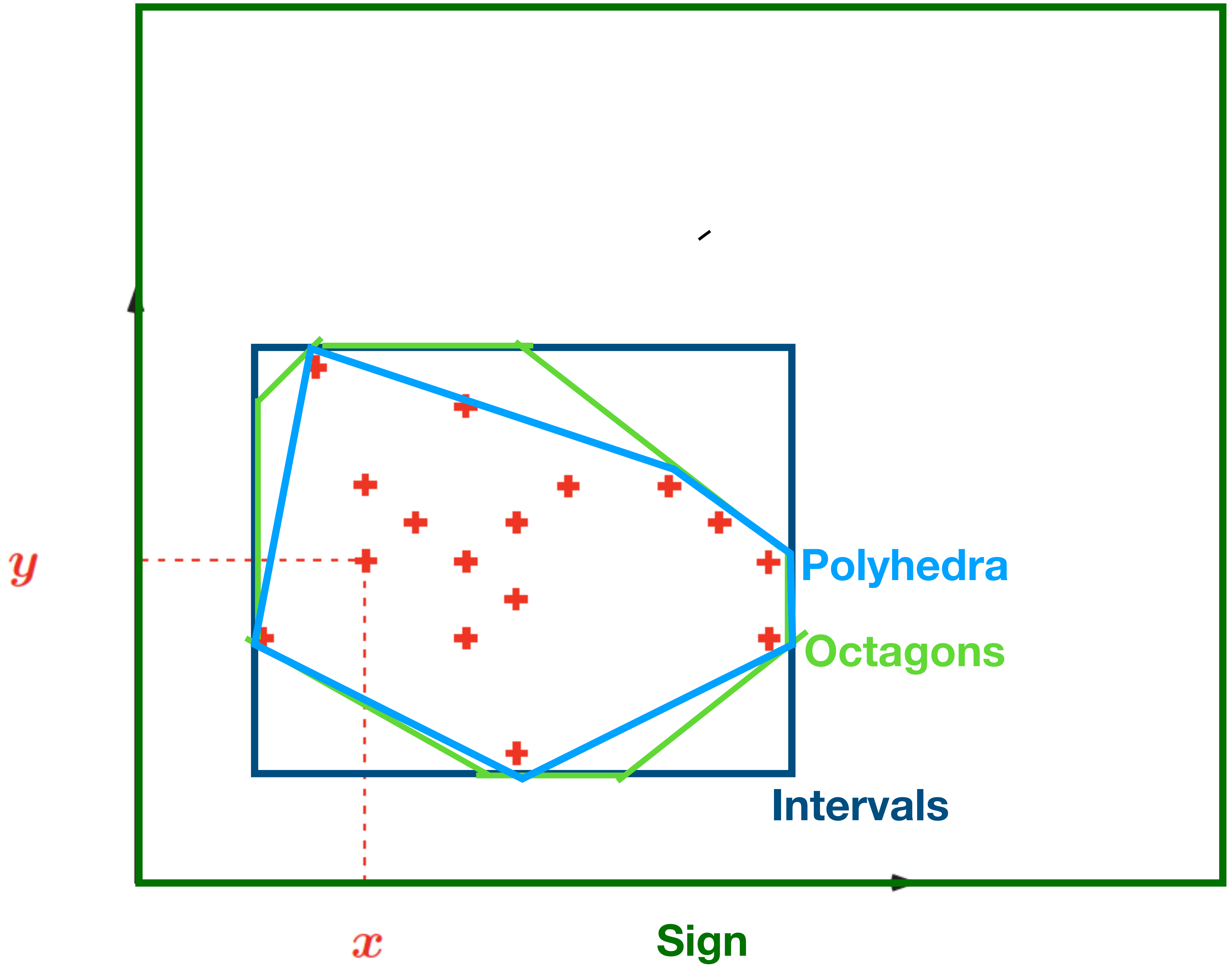
$$\{\ldots(19{,}77)\ldots(20{,}03)\ldots.\}$$



Refinements of abstraction

An (in)-finite set of points :

$\{\ldots(19{,}77)\ldots(20{,}03)\ldots.\}$



Order on abstract domains

We say that the abstract domain A_1 refines A_2,

written $A_1 \preceq A_2$, iff

$$\forall c \in C \,.\, \gamma_{A_1}(\alpha_{A_1}(c)) \subseteq \gamma_{A_2}(\alpha_{A_2}(c))$$

intuitively, A_1 is more precise than A_2

$$Octagons \sqsubseteq Int \sqsubseteq Sign$$

Conjunctive properties

program verification often requires the use of the conjunction of several basic predicates

concrete states = stores with two variables x, y

intervals abstraction for each variable

abstract state = an interval for each variable

`[0,∞] [3,8]`

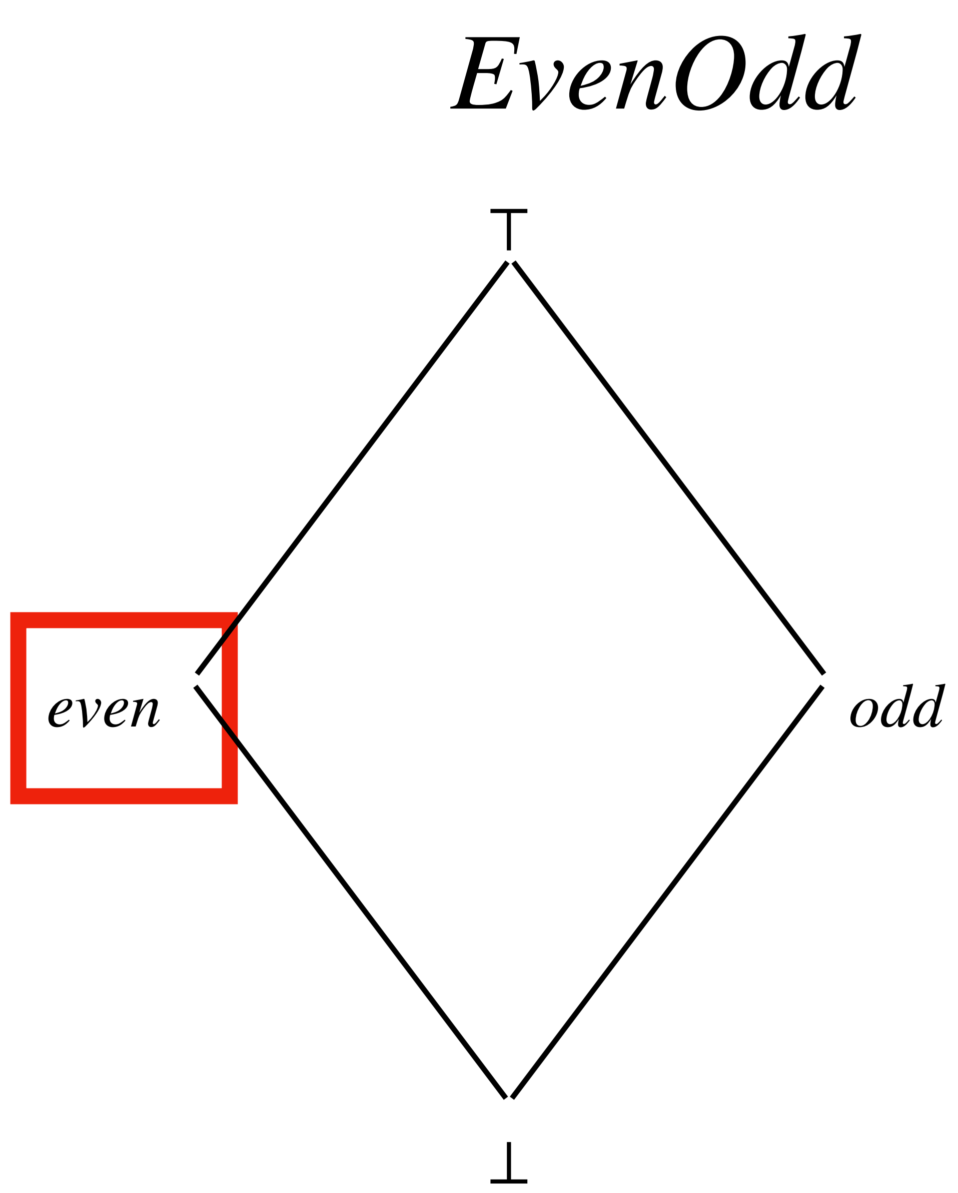
Product domain

$$C \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} A_0 \qquad\qquad C \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$$

$$C \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} A_0 \times A_1$$

$$\gamma_\times(a_0, a_1) = \gamma_0(a_0) \cap \gamma_1(a_1)$$

Problem



concrete stores = stores with one variable x

Int × EvenOdd

e.g. an abstract state $([2,10], \mathit{even})$

describes **even** values between 2 and 10

but also $([1,11], \mathit{even})$ represents the same

concrete set $\{2,4,6,8,10\}$!

Reduced product $A_0 \sqcap A_1$

$$C \underset{\alpha_0}{\overset{\gamma_0}{\leftrightarrows}} A_0 \qquad\qquad C \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} A_1$$

$$C \underset{\alpha_\sqcap}{\overset{\gamma_\sqcap}{\leftrightarrows}} (A_0 \times A_1)_\equiv \quad A_0 \sqcap A_1$$

take the equivalence classes

$$(a_0, a_1) \equiv (a_0', a_1') \Leftrightarrow \gamma_\times(a_0, a_1) = \gamma_\times(a_0', a_1')$$

$$\gamma_\sqcap([a_0, a_1]_\equiv) = \gamma_0(a_0) \cap \gamma_1(a_1)$$

Abstract program analysis

Regular commands

regular command

atomic command

$$
\begin{aligned}
r ::= \;& e \\
\mid \;& r_1 ; r_2 \\
\mid \;& r_1 + r_2 \\
\mid \;& r^\star
\end{aligned}
$$

choice

Kleene star

$$e ::= \mathsf{skip} \mid x := a \mid b? \mid \ldots$$

$$a ::= n \mid x \mid a_1 + a_2 \mid \ldots$$

$$b ::= a_1 \leq a_2 \mid b_1 \wedge b_2 \mid \ldots$$

Collecting semantics

$$[\![\mathsf{skip}]\!]P \triangleq P$$

$$[\![x := a]\!]P \triangleq \{\sigma[x \mapsto [\![a]\!]\sigma] \mid \sigma \in P\}$$

$$[\![b?]\!]P \triangleq [\![b]\!]P$$

$$[\![r_1; r_2]\!]P \triangleq [\![r_2]\!]([\![r_1]\!]P)$$

$$[\![r_1 + r_2]\!]P \triangleq [\![r_1]\!]P \cup [\![r_2]\!]P$$

$$[\![r^\star]\!]P \triangleq \bigcup_{k=0}^{\infty} [\![r]\!]^k P$$

Abstract semantics

$$[\![e]\!]^{\#}_{A} a \triangleq [\![e]\!]^{A} \triangleq (\alpha \circ [\![e]\!] \circ \gamma) a$$

$$[\![r_1 ; r_2]\!]^{\#}_{A} a \triangleq [\![r_2]\!]^{\#}_{A}([\![r_1]\!]^{\#}_{A} a)$$

Just a composition of bcas!

$$[\![r_1 + r_2]\!]^{\#}_{A} a \triangleq [\![r_1]\!]^{\#}_{A} a \vee [\![r_2]\!]^{\#}_{A} a$$

$$[\![r^{\star}]\!]^{\#}_{A} a \triangleq \bigvee_{k=0}^{\infty} ([\![r]\!]^{\#}_{A})^{k} a$$

Example on Interval

c_1

x := 10;
while (x>0) {
    x := x-1
}; **{ x = 0 }?**

Complete!

$[\,x \mapsto \top\,]$

$x := 10;$

$(\quad [\,x \mapsto [0,10]\,]$

$\quad (x > 0)?;$

$\quad [\,x \mapsto [1,10]\,]$ ← Abstract loop invariant

$\quad x := x - 1;$

$\quad [\,x \mapsto [0,9]\,]$

$)^*; [\,x \mapsto [0,10]\,]$

$(x \leq 0)?$

$[\,x \mapsto [0,0]\,]$

Example on Interval

c_2

```
x := 10;
while (x>1) {
      x := x-2
   };
```

{ x = 0 }?

Incomplete!

$[\mathtt{x} \mapsto \top]$

$x := 10;$

$(\quad [\mathtt{x} \mapsto [0,10]]$

$\quad (x > 1)?;$

$\quad [\mathtt{x} \mapsto [2,10]]$

Abstract loop invariant

$\quad x := x - 2;$

$\quad [\mathtt{x} \mapsto [0,8]]$

$)^*; \quad [\mathtt{x} \mapsto [0,10]]$

$(x \leq 1)?$

$[\mathtt{x} \mapsto [0,1]]$

The precision of the analysis depends on how the program is written!!!

Complete!

Incomplete!

Like complexity is a property of the program not of the computed function!!

c_1

```
x := 10;
while (x>0) {
        x := x-1
    }; { x = 0 }
```

$x \in [0,0]$

c_2

```
x := 10;
while (x>1) {
        x := x-2
    }; { x = 0 }
```

$x \in [0,1]$

LICS 2021

A Logic for Locally Complete Abstract Interpretations

Roberto Bruni*, Roberto Giacobazzi‡, Roberta Gori*, Francesco Ranzato§
*University of Pisa, Italy
‡University of Verona, Italy
§University of Padova, Italy

In loving memory of Anna Maria De Paolis and Dina Gorini

2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) | 978-1-6654-4895-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/LICS52264.2021.9470608

***Abstract*—We introduce the notion of *local completeness* in abstract interpretation and define a logic for proving both the correctness and incorrectness of some program specification. Abstract interpretation is extensively used to design sound-by-construction program analyses that over-approximate program behaviours. Completeness of an abstract interpretation A for all possible programs and inputs would be an ideal situation for verifying correctness specifications, because the analysis can be done compositionally and no false alert will arise. Our first result shows that the class of programs whose abstract analysis on A is complete for all inputs has a severely limited expressiveness. A novel notion of *local completeness* weakens the above requirements by considering only some specific, rather than all, program inputs and thus finds wider applicability. In fact, our main contribution is the design of a proof system, parameterized by an abstraction A, that, for the first time, combines over- and under-approximations of program behaviours. Thanks to local completeness, in a provable triple $\vdash_A [P]\ \mathsf{c}\ [Q]$, the assertion Q is an under-approximation of the strongest post-condition $\mathsf{post}[\mathsf{c}](P)$ such that the abstractions in A of Q and $\mathsf{post}[\mathsf{c}](P)$ coincide. This means that Q is never too coarse, namely, under mild assumptions, *the abstract interpretation of* c *does not yield false alerts for the input* P *iff* Q *has no alert*. Thus, $\vdash_A [P]\ \mathsf{c}\ [Q]$ not only ensures that all the alerts raised in Q are true ones, but also that if Q does not raise alerts then c is correct.**

I. Introduction

Technology, you can't live without. But any coin has two sides and software failures are increasingly more frequent and their consequences are more disruptive in the Digital Age than ever before. Quoting Dijkstra's speech at the Turing Award lecture [11], *the only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness*. Since correctness proof attempts may fail even when the program is correct, also incorrectness proofs would be needed to catch actual bugs, because you can't fix what you can't see. Code-review processes and test-driven development are widely adopted best practices in software companies. Nevertheless, the problem is far from being solved and static reasoning should be extended to bug catching, as advocated by O'Hearn's incorrectness logic (IL) [24].

Static program analysis has been investigated and used for over half century and is a major methodology to help programmers and software engineers in producing reliable code [4], [12], [15], [18], [23], [27], [28]. Static analysis is based on symbolic reasoning techniques to prove program properties without running them. Given a program c and a correctness specification Spec, the aim of a static verification is either to prove that the behaviour of c satisfies Spec or to raise some alerts that point out which circumstances *may* cause a violation of Spec. The conditional is needed because, starting with the fundamental works by Hoare [18], program verifiers tend to over-approximate the program behaviour: this is an unavoidable consequence of the will to solve an otherwise undecidable analysis problem. As any alerting system, program analysis turns out to be *credible*, when few, ideally zero, false alerts are reported to the user [9]. The dual perspective has been recently tackled by incorrectness logic [24]: exploiting under-approximations, any violation exposed by the analysis is a true alert. This makes IL a credible support for code-review, but Spec may be violated even when no alert is reported.

Abstract interpretation [6]–[8] is a well-established framework for designing sound-by-construction over-approximations of the program behaviour. Given an abstraction A, instead of verifying whether the strongest post-condition $\mathsf{post}[\mathsf{c}](P)$ for a program c and a pre-condition P (also written $[\![\mathsf{c}]\!]P$) satisfies a correctness specification Spec, a (sound) abstract over-approximation $A(\mathsf{post}[\mathsf{c}](P))$ is considered. While it is obvious that if $A(\mathsf{post}[\mathsf{c}](P))$ satisfies Spec then the program is correct, it may happen that $A(\mathsf{post}[\mathsf{c}](P))$ does not satisfy Spec even if the program is correct, because A introduced false alerts. Once the specification Spec and its abstract approximation in A coincide, the ideal program analysis is achieved by assuring that a sound analysis is also *complete*, so that no false alert is ever raised.

Technically, in a domain A of abstract program stores, with abstraction and concretization maps α and γ resp., any store property P is, in general, over-approximated by $A(P) = \gamma\alpha(P) \supseteq P$. Assuming that Spec is expressible in A means that $\mathit{Spec} = A(\mathit{Spec})$ holds. For instance, in the abstract domain of intervals Int (see Example III.5) the property $x \geq 0$ is expressible by the infinite interval $[0, +\infty]$. By contrast, $x \neq 0$ is not expressible in Int, since the least over-approximating interval is $\mathsf{Int}(x \neq 0) = \mathbb{Z} \supsetneq \mathbb{Z} \smallsetminus \{0\}$. An abstract semantics associates with each program c a computable function $\mathsf{post}_A[\mathsf{c}] : A \to A$ on the abstraction A (also written $[\![\mathsf{c}]\!]^\sharp_A$). By soundness of abstract interpretation, if $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \subseteq \mathit{Spec}$ then $\{P\}\ \mathsf{c}\ \{\mathit{Spec}\}$ is a valid Hoare triple. However, when $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \not\subseteq \mathit{Spec}$ we cannot conclude that $\{P\}\ \mathsf{c}\ \{\mathit{Spec}\}$ is not valid, because any witness in $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \smallsetminus \mathit{Spec}$ is just a *potentially false alert*.

978-1-6654-4895-6/21/$31.00 ©2021 IEEE

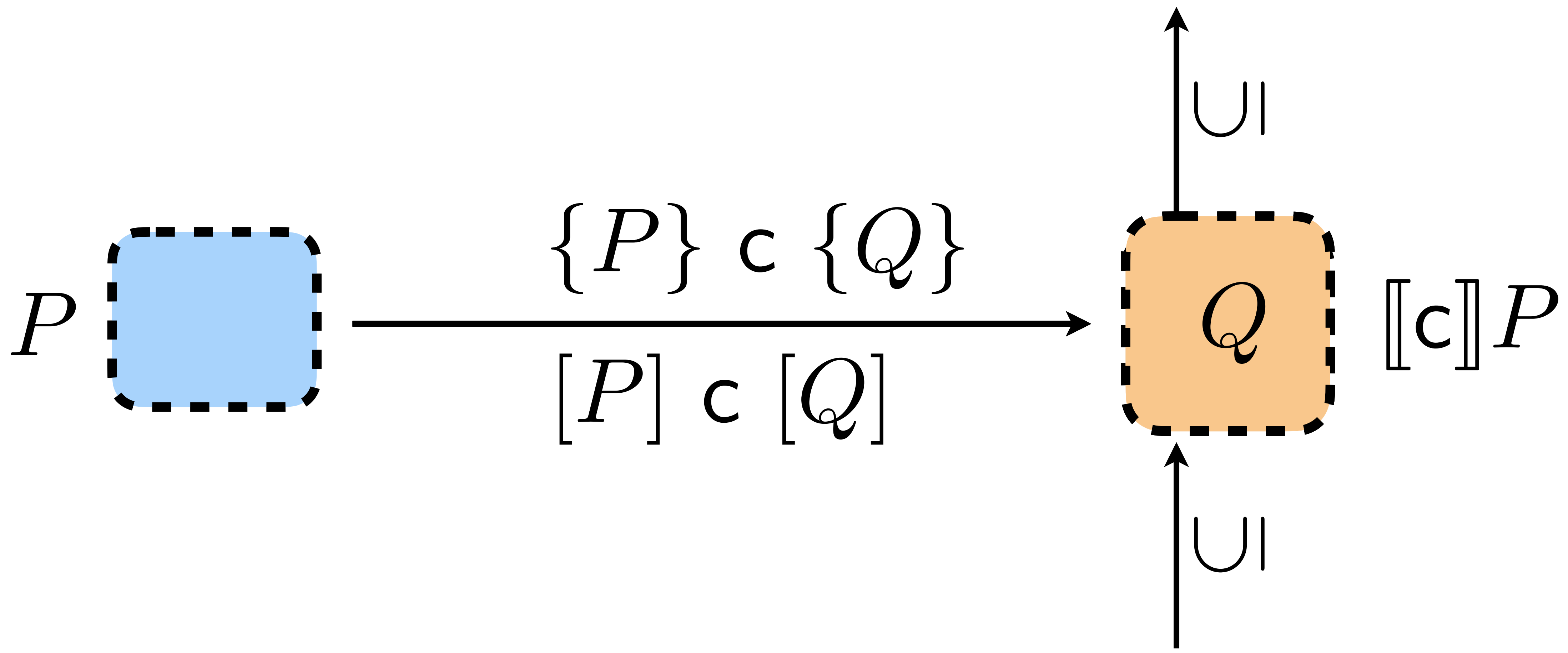
any (locally) complete under approximation either proves the program correct or incorrect (without false positives)

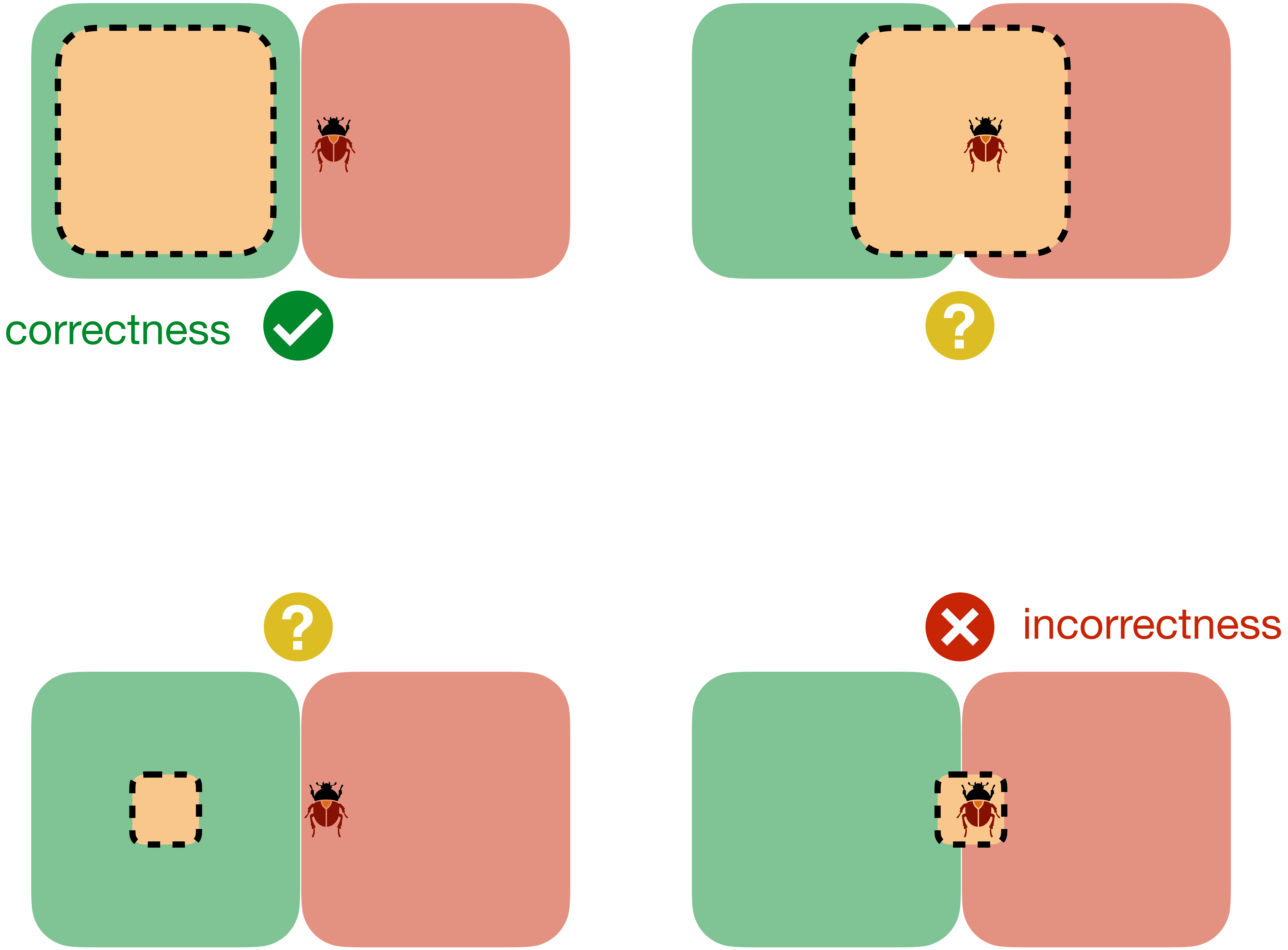
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and

Over vs Under

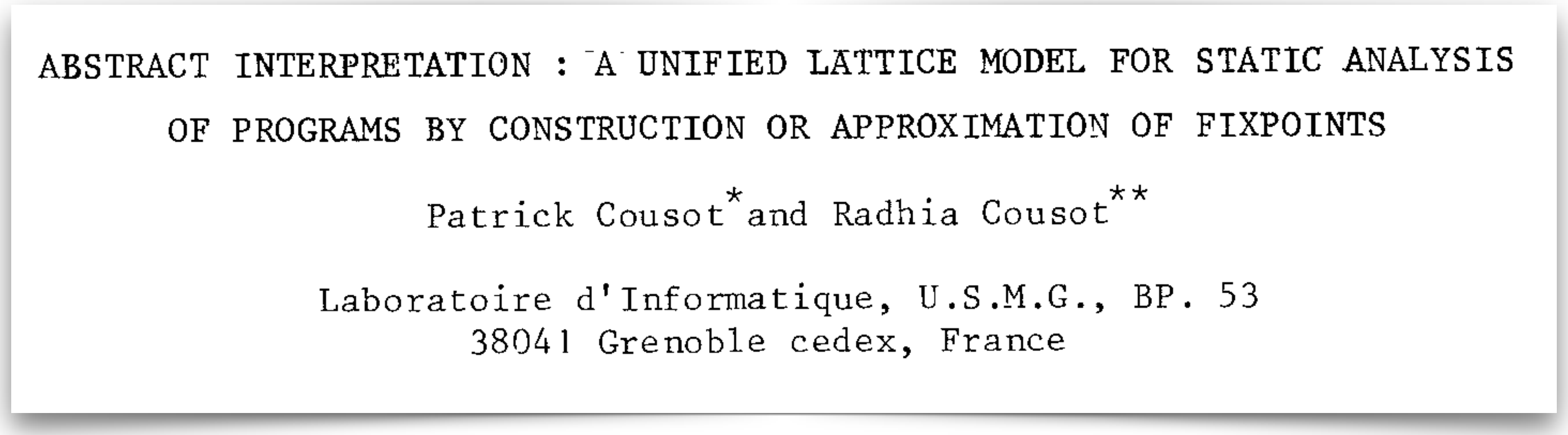




Incorrectness Logic

PETER W. O'HEARN, Facebook and University College London, UK

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare's logic of correctness.

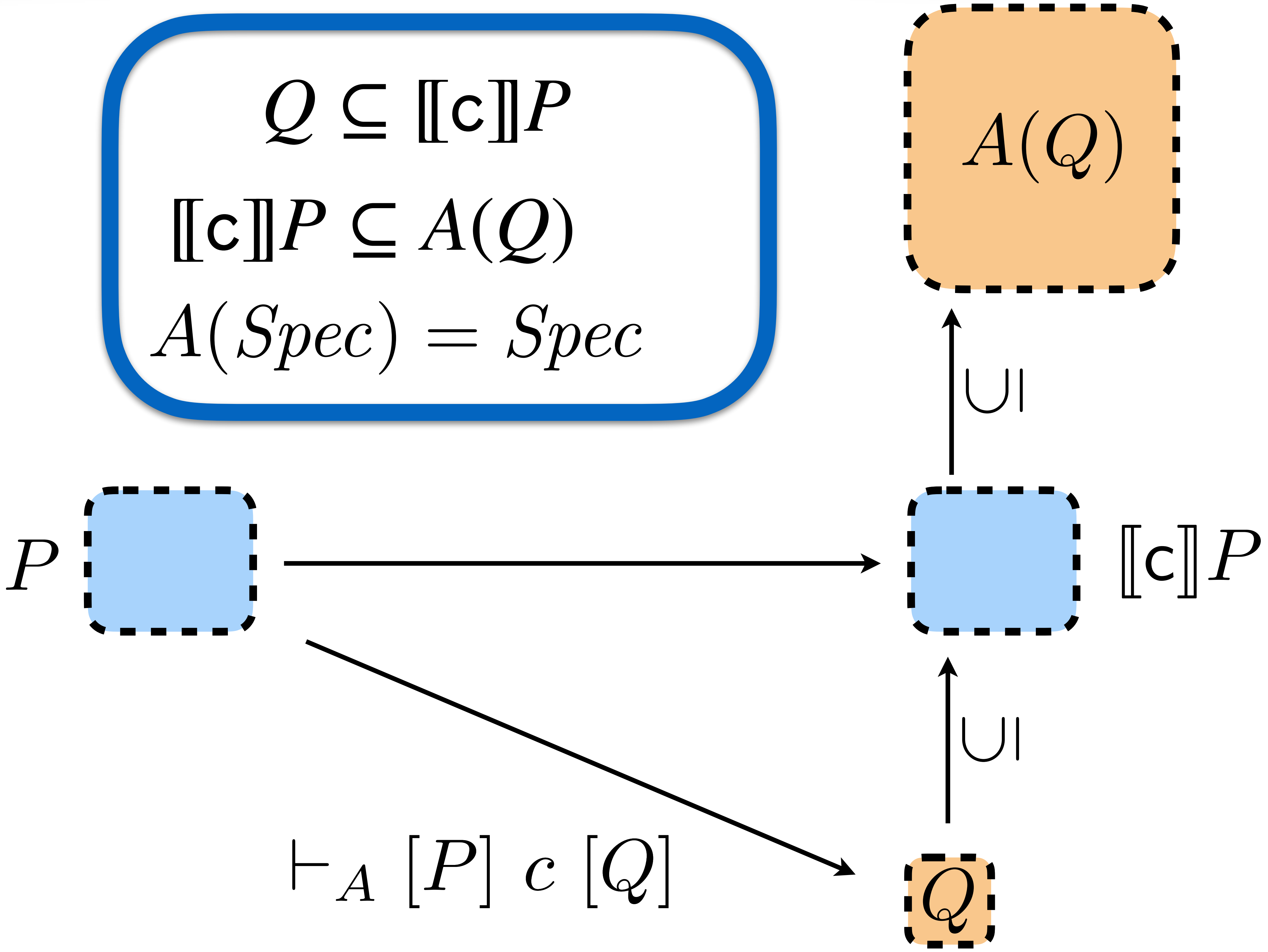

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

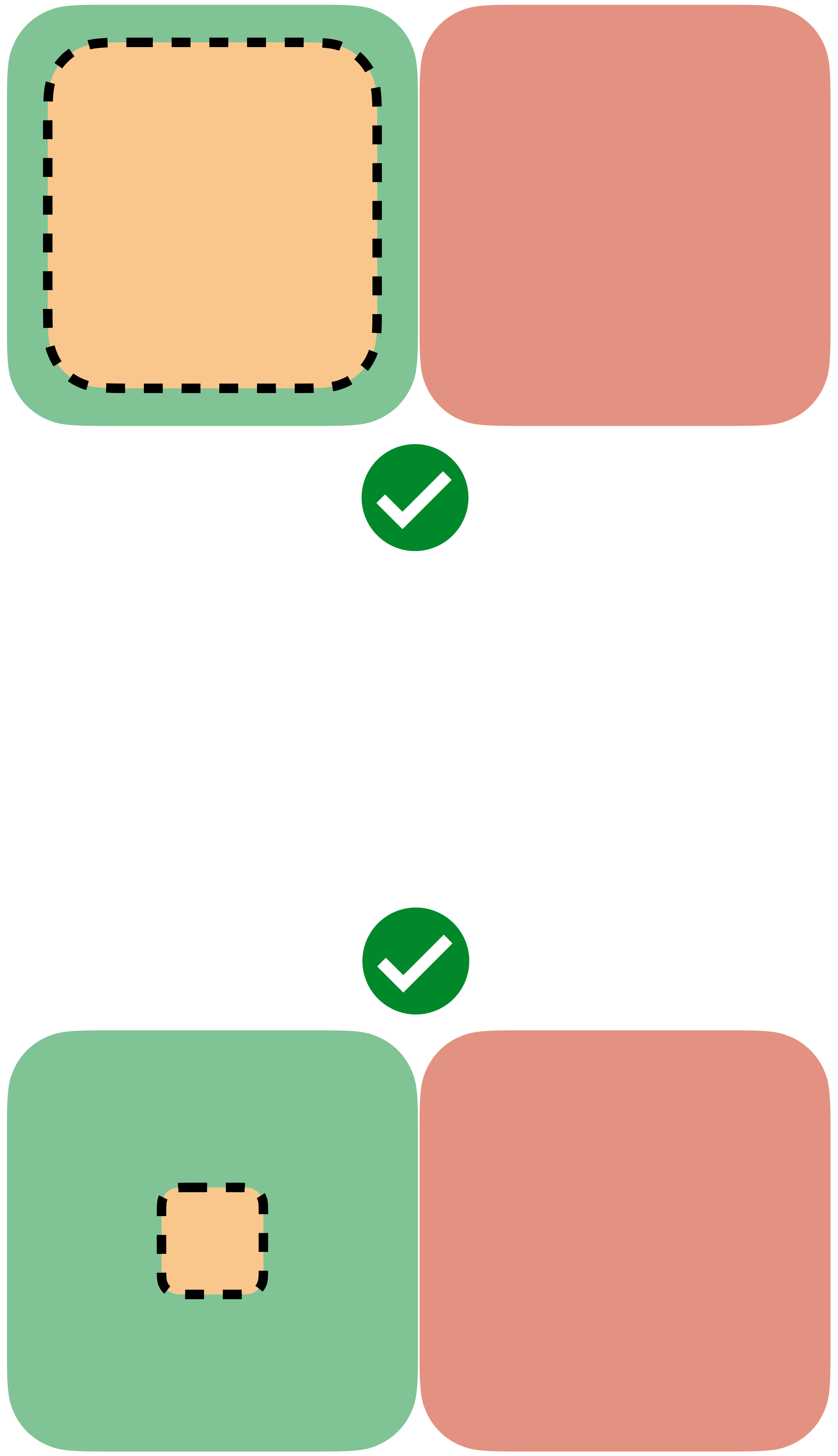
Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

The idea

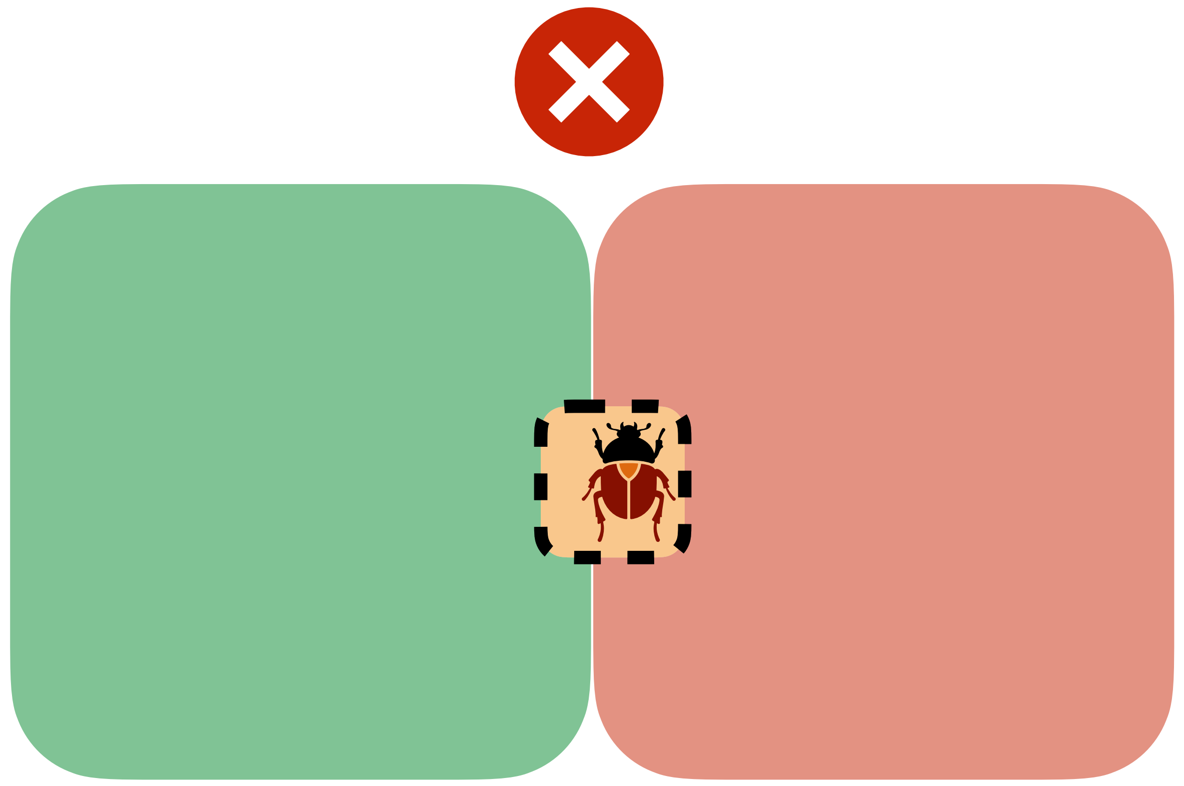
$$Q \subseteq [\![\mathsf{c}]\!]P$$
$$[\![\mathsf{c}]\!]P \subseteq A(Q)$$
$$A(Spec) = Spec$$





Local completeness

$$A(Q) \subseteq Spec$$
$$\Leftrightarrow$$
$$[\![\mathsf{c}]\!]P \subseteq Spec$$
$$\Leftrightarrow$$
$$Q \subseteq Spec$$



O'Hearn's triples

pre condition

$$[P]\ c\ [Q]$$

post condition

any output matching the postcondition
can be reached by executing the command
on some input matching the precondition

$$[\![c]\!]P \supseteq Q$$

under approximation!

includes just reachable states

LCL triples

pre condition

$$\vdash_A [P]\ c\ [Q]$$

abstract domain

post condition

any output matching the postcondition
can be reached by executing the command
on some input matching the precondition
+
for any input matching the precondition executing the command establishes the abstraction of the postcondition

can include non reachable states

$$A(Q) \supseteq [\![c]\!]P \supseteq Q$$

over approximation!

under approximation!

includes just reachable states

Combining under and over approximations

Logical correctness

Th.

If $\vdash_A [P]\ r\ [Q]$ then $Q \subseteq [\![r]\!]P \subseteq A(Q) = [\![r]\!]^{\#}_A A(P)$

Proof.

By induction on the derivation.

Verification

Th.

If $A(Spec) = Spec$, then any provable triple $\vdash_A [P]\ r\ [Q]$

either shows the program correct ($Q \subseteq Spec$)

or exposes some true positives ($Q \setminus Spec \neq \emptyset$)

correctness + bug finding!

Proof.

$$
\begin{aligned}
[\![r]\!]P \subseteq Spec &\Leftrightarrow A[\![r]\!]P \subseteq Spec \\
&\Leftrightarrow A(Q) \subseteq Spec \\
&\Leftrightarrow Q \subseteq Spec
\end{aligned}
$$

Questions

Question 1

Let $P \triangleq (x \in \{-7,5\})$ and $r \triangleq (x < 0)?; x := -x$.

1. Compute the abstract semantics $[\![c]\!]^{\#}_{\mathsf{Sign}}$ on $\alpha_{\mathsf{Sign}}(P)$
2. Check if the result is the same as $\alpha_{\mathsf{Sign}}([\![c]\!]P)$

Question 2

What is the bca for the test (=0?) in the Interval domain?

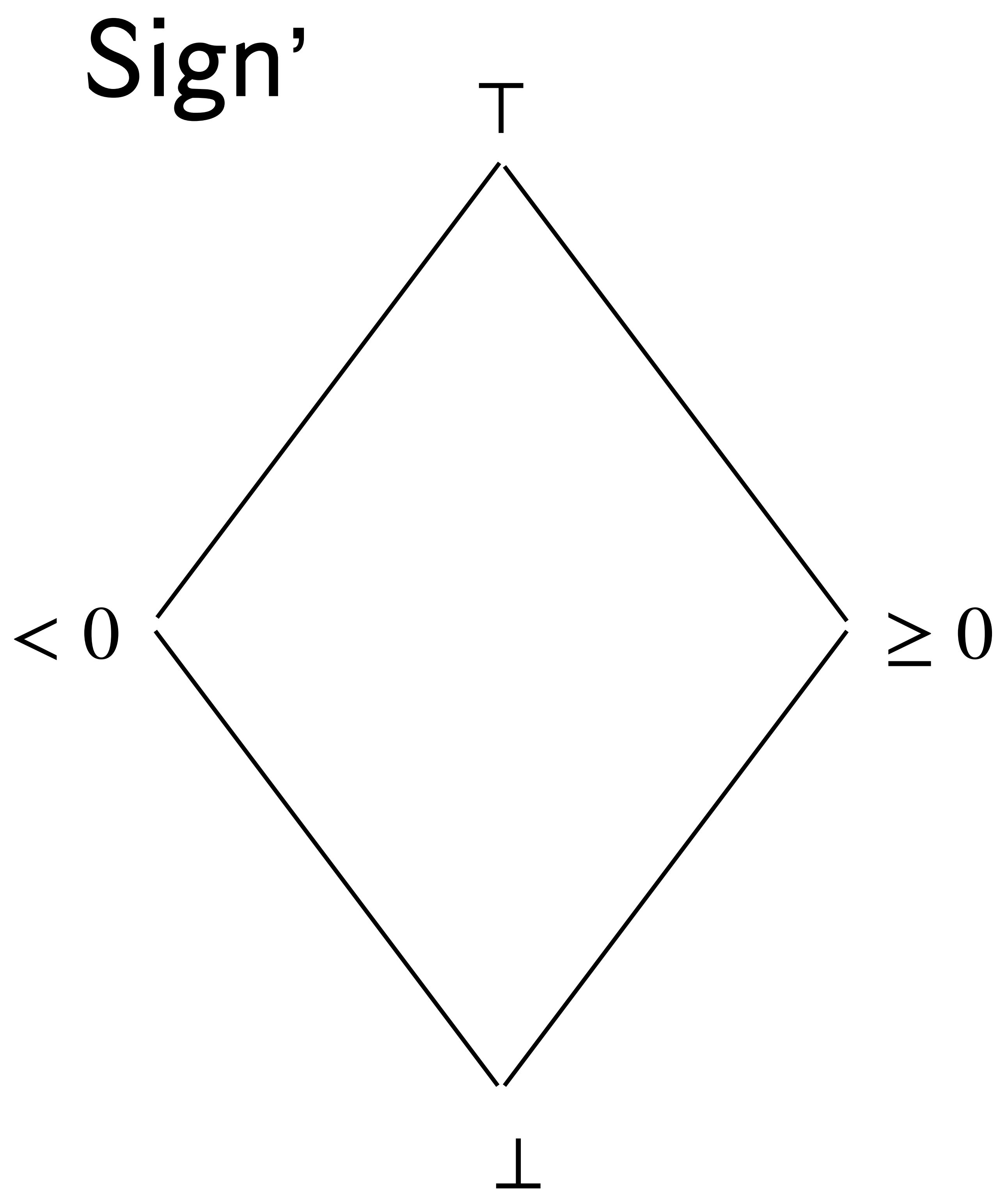
$$(=0?)^{\text{Int}}[n, m] = \begin{cases} [0,0] & \text{If } n \leq 0 \leq m \\ \bot & \text{Otherwise} \end{cases}$$

Exam question

Consider the abstract domain Sign' in the figure

1. Define the corresponding α and γ.

2. Does it admit a complete abstract multiplication?



Take-home message

Different approaches are often seen in opposition one each other but we could gain much more from their combination:
abandon any preconception, be open minded!

stop fighting: each approach has its own merit, none is better than the others {many} (HL ; NC ; SL ; IL ; ISL

+

SIL ; SepSIL ; AI)* [thanks]