HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

Abstract Interpretation

CS252r Fall 2015

Slides from Jeff Foster, in turn based on lectures by David Schmidt and Alex Aiken

Also refers to slides from Principles of Program Analysis by Nielson, Nielson, and Hankin

http://www2.imm.dtu.dk/~riis/PPA/ppasup2004.html

What is an Abstraction?

- An abstraction is a property from some domain

Elephant (species)

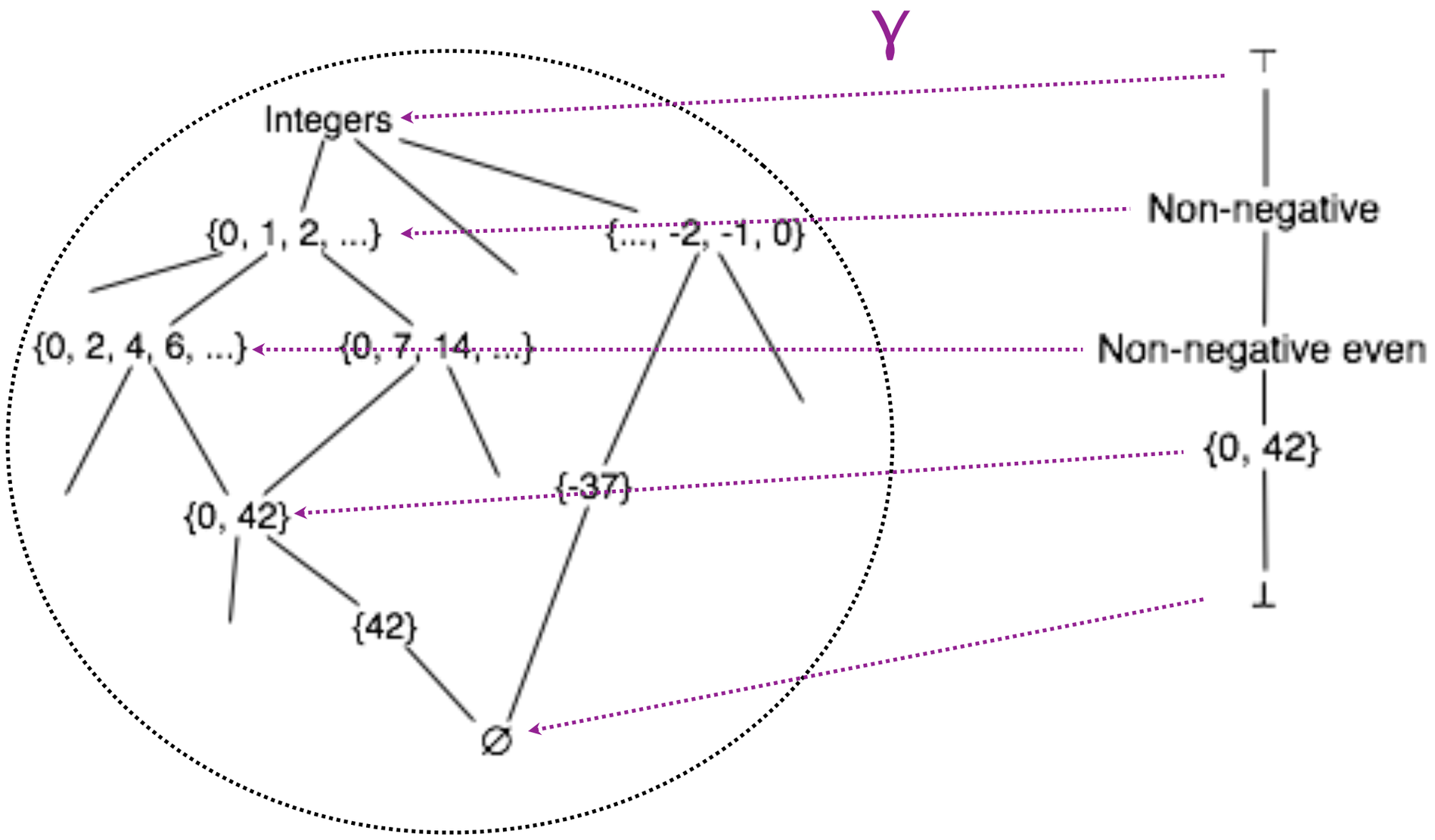
Grey (color)

Heavy (weight)

⊔ll

4000..6000kg

Example Abstraction

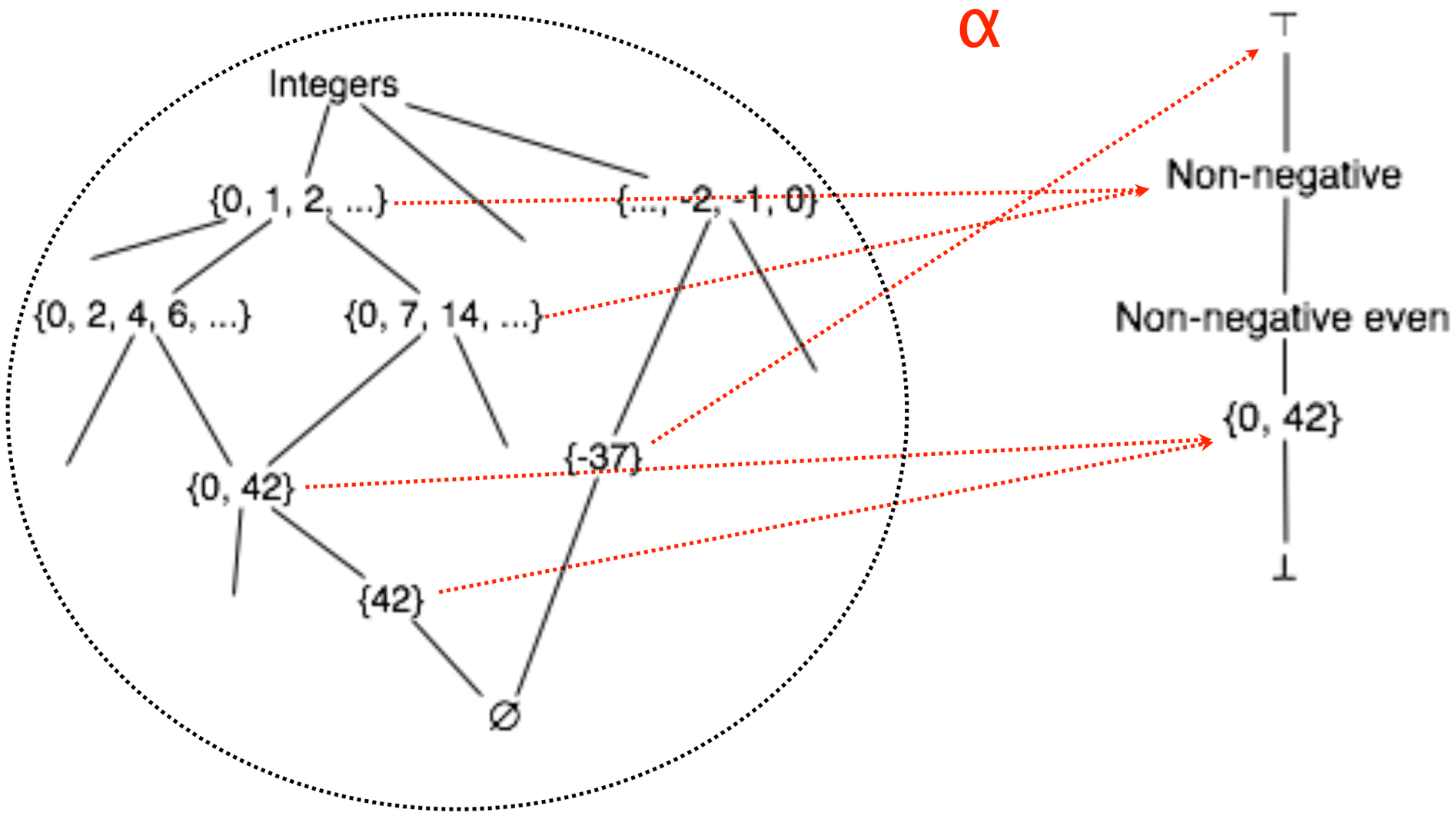


Concrete values: sets of integers

Abstract values

Concretization function γ maps each abstract value to concrete values it represents

Abstraction is Imprecise

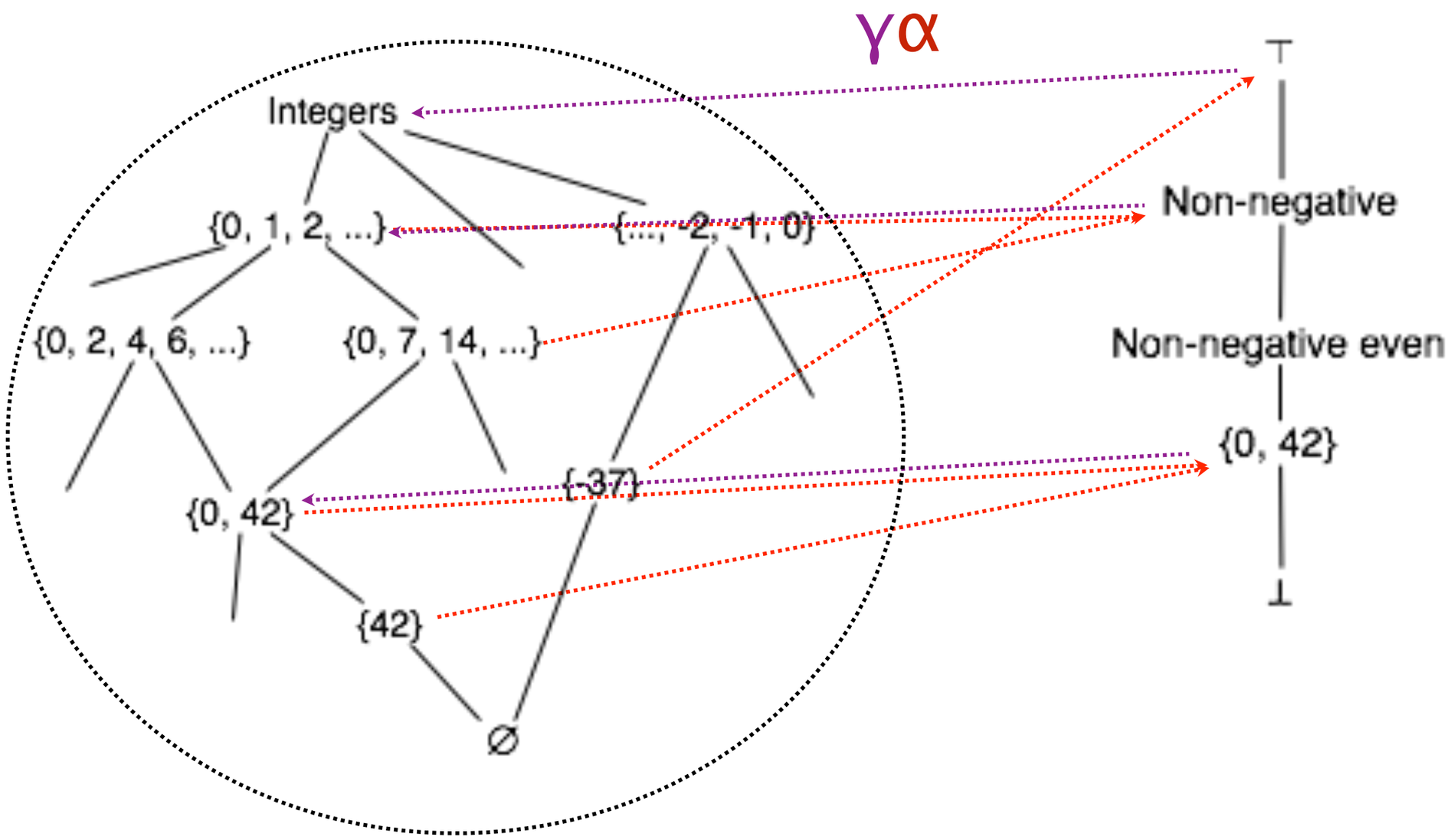


Concrete values: sets of integers

Abstract values

Abstraction function α maps each concrete set to the best abstract value

Composing α and γ



Concrete values: sets of integers

Abstract values

Abstraction followed by concretization is sound but imprecise

α and γ Form a Galois Insertion

- α and γ form a Galois insertion iff
  - α and γ are monotonic
    - Recall: f is monotonic if $x \leq y \Rightarrow f(x) \leq f(y)$
  - $S \subseteq \gamma(\alpha(S))$ for any concrete set S
  - $\alpha(\gamma(A)) = A$ for any abstract element A
- Next up: Abstract interpretation in action
  - We'll develop an abstract interpretation of a simple language and prove it correct using these ideas

Source Language

- Integers and multiplication
  - e ::= i | e * e
- Standard semantics of the program
  - Eval : e → Int
  - Eval(i) = i
  - Eval(e1*e2) = Eval(e1) × Eval(e2)

Abstraction

- Define an abstract semantics that computes only the sign of the result

×	+	0	-
+	+	0	-
0	0	0	0
-	-	0	+

AEval : e → {-, 0, +}

$$\text{AEval}(i) = \begin{cases} + & i > 0 \\ 0 & i = 0 \\ - & i < 0 \end{cases}$$

AEval(e1*e2) = AEval(e1) × AEval(e2)

Soundness

- We can show our abstraction correctly predicts the sign of an expression
- Proof: by structural induction on e
  - Eval(e) > 0 iff AEval(e) = +
  - Eval(e) = 0 iff AEval(e) = 0
  - Eval(e) < 0 iff AEval(e) = -

Another Approach to Soundness

- Natural concretization function

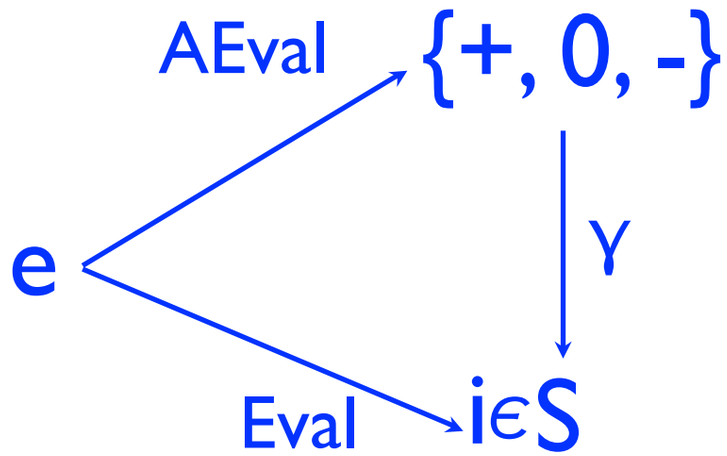
$$\gamma(+) = \{i \mid i>0\}$$

$$\gamma(0) = \{0\}$$

$$\gamma(-) = \{i \mid i<0\}$$

- Note: This presentation is slightly non-standard
  - Usually defined in terms of execution traces

Soundness (cont'd)



- Our abstraction is sound if
  - Eval(e) ∈ γ(AEval(e))
- Soundness proof: later

Adding Unary Negation

- e ::= i | e * e | -e

- Eval(-e) = -Eval(e)

- AEval(e) = -AEval(e)

=	+	0	-
	-	0	+

- No problems

Adding Addition

- e ::= i | e * e | -e | e + e

- Eval(e1+e2) = Eval(e1) + Eval(e2)

- AEval(e1+e2) = AEval(e1) + AEval(e2)

±	+	0	-
+	+	+	?
0	+	0	-
-	?	-	-

Our abstract domain is not closed under addition

Solution

- Add an abstract value to represent any integer
- Finding closed domain often key design problem

$\gamma(\top) = \{\text{integers}\}$

$\pm$	+	0	-	$\top$
+	+	+	$\top$	$\top$
0	+	0	-	$\top$
-	$\top$	-	-	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

- Other operations also need to handle $\top$

Two Ways to Lose Information

- OK: Abstraction still precise enough
  - Eval((5 * 5) + 6) = 31
  - AEval((5*5) + 6) = (+ $\underline{\times}$ +) $\underline{+}$ + = +
    - Abstractly, we don't know which value we computed
    - ...but we don't care, since we only want the sign

- Not so good: "Don't know" values
  - Eval((1 + 2) + -3) = 0
  - AEval((1 + 2) + -3) = (+ $\underline{+}$ +) $\underline{+}$ - = + $\underline{+}$ - = $\top$
    - We also don't know which value we computed
    - ...and we can't even figure out its sign

Adding Integer Division

- What happens when we divide by zero?
  - The result is not an integer (it's undefined)
  - If we divide each integer in a set by 0, the result is the empty set

$$\gamma(\perp) = \varnothing$$

$\div$	+	0	-	$\top$	$\perp$
+	+	0	-	$\top$	$\perp$
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
-	-	0	+	$\top$	$\perp$
$\top$	$\top$	0	$\top$	$\top$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

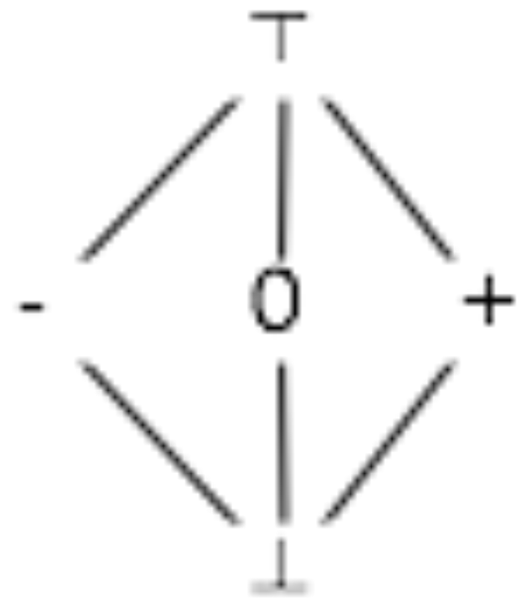
Adding Integer Division (cont'd)

- We need to extend other abstract operations to work on ⊥
- Every operation involving ⊥ results in ⊥
  - All operations are strict in ⊥

$$\begin{aligned} \bot \times a &= \bot \\ a \times \bot &= \bot \\ \bot + a &= \bot \\ a + \bot &= \bot \\ -\bot &= \bot \end{aligned}$$

The Abstract Domain

- Look, Ma, a lattice!
- We've got:
  - A set of elements {⊥, +, 0, -, ⊤}
  - A relation < that is
    - Reflexive
    - Anti-symmetric
    - Transitive
  - And
    - The least upper bound (lub, ⊔) and greatest lower bound (glb, ⊓) exists for any pair of elements
    - So it's a lattice



Abstraction and Concretization

- Concretization function γ

$$\gamma(\top) = \text{all integers}$$
$$\gamma(+) = \{i \mid i>0\}$$
$$\gamma(0) = \{0\}$$
$$\gamma(-) = \{i \mid i<0\}$$
$$\gamma(\bot) = \varnothing$$

- Abstraction function maps concrete values (sets of integers) to smallest valid abstract element
  - $\alpha(S) = \left\{ \begin{array}{ll} - & \exists i \in S \,.\, i<0 \\ \bot & \text{otherwise} \end{array} \right\} \sqcup \left\{ \begin{array}{ll} 0 & \exists i \in S \,.\, i=0 \\ \bot & \text{otherwise} \end{array} \right\} \sqcup \left\{ \begin{array}{ll} + & \exists i \in S \,.\, i>0 \\ \bot & \text{otherwise} \end{array} \right\}$

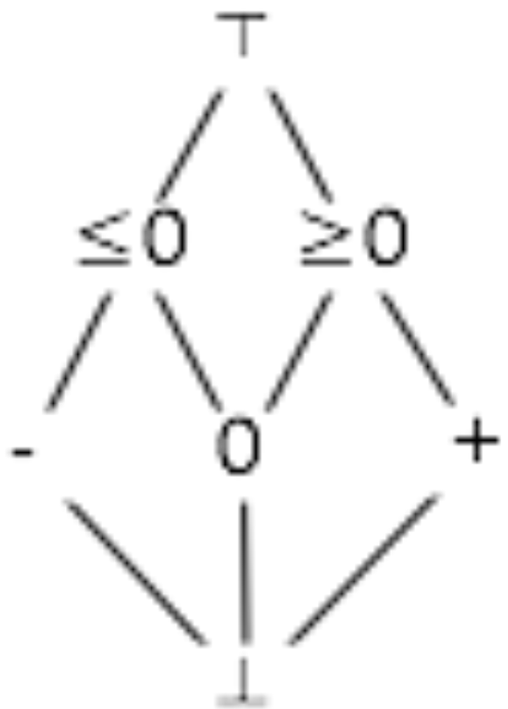
Definition

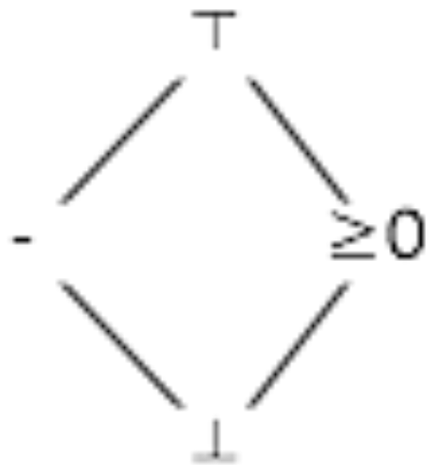
- An abstract interpretation consists of
  - A concrete domain S and an abstract domain A
  - Concretization and abstraction functions that form a Galois insertion [of A into S]
  - A (sound) abstract semantic function

- Recall: α and γ form a Galois insertion if
  - α and γ are monotone
  - $S \subseteq \gamma(\alpha(S))$ or $id \leq \gamma\alpha$ for any concrete set S
  - $A=\alpha(\gamma(A))$ or $id = \alpha\gamma$ for any abstract element A

An Alternate Abstract Domain

- That domain wasn't the only choice, of course





…

- The right domain depends on the problem we're trying to solve

Relationship to Data Flow Analysis

- Abstract interpretation was invented partially to find a firm semantic foundation for data flow analysis
  - Precise relationship between concrete domain (program executions) and abstract domain (data flow facts)
  - Generic correctness proof

- Caveat: Data flow typically uses meet, abstract interpretation typically uses join

Conclusions

- Cousot and Cousot paper(s) seminal work(s)
- The theory of abstract interpretation is often confused with using it to construct tool (e.g., data flow analysis)

- Slogan:
  - Finite lattices + monotonic functions = program analysis

Reference

- Slides 2-16, 37-58 from Chap. 4 of Principles of Program Analysis by Nielson, Nielson, and Hankin http://www.imm.dtu.dk/~riis/PPA/slides4.pdf