All the world is an abstract interpretation (of all the world)

David Schmidt

Kansas State University

An abstraction is a property from some domain

brown **(color)**

.

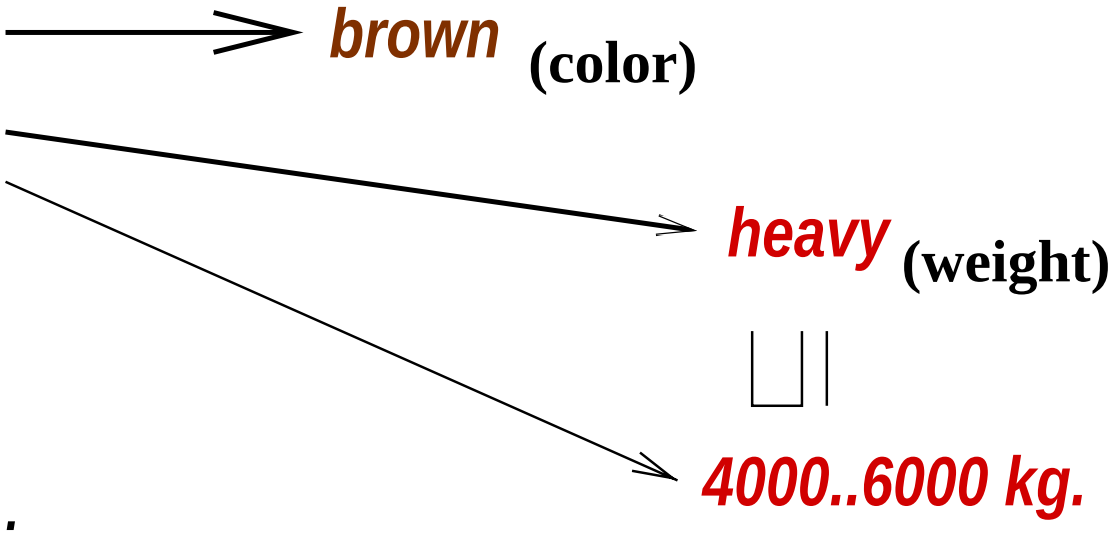
An abstraction is a property (cont.)

brown **(color)**

heavy **(weight)**

.

An abstraction is a property (cont.)



.

An abstraction is a property (concl.)

elephant (species)

brown (color)

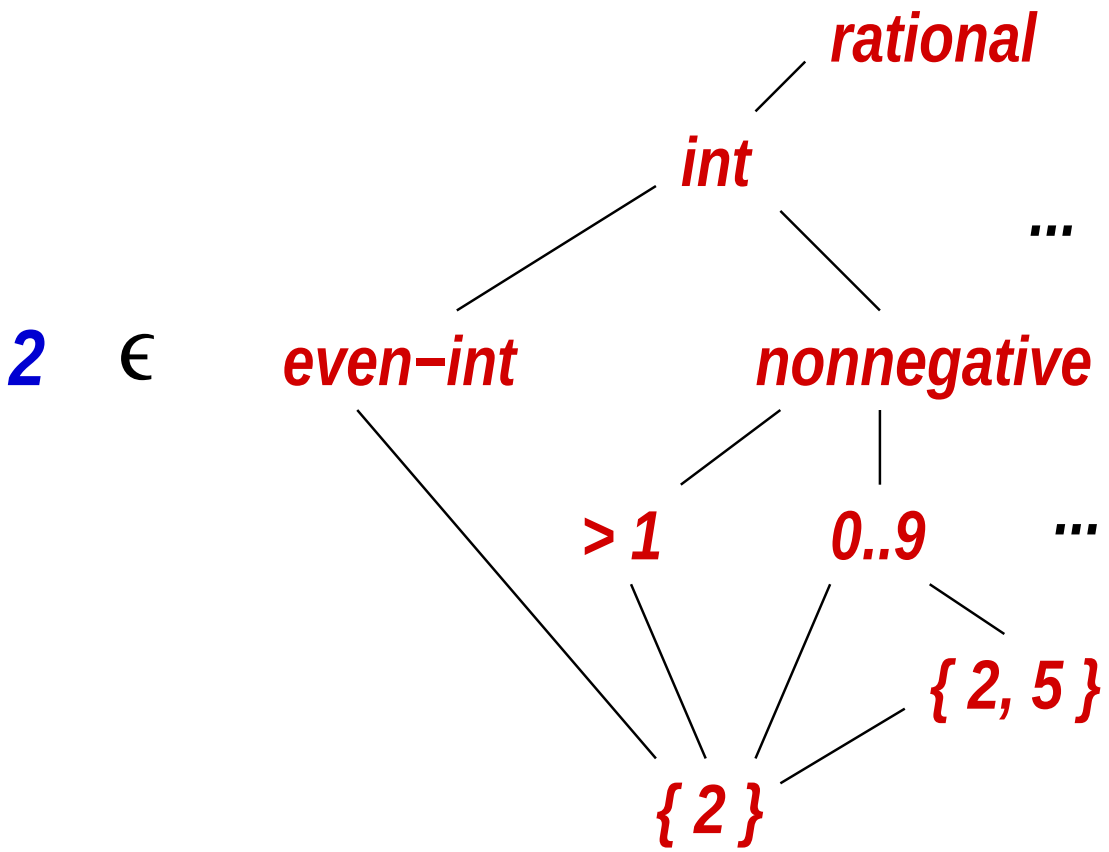
heavy (weight)

$\sqcup$

4000..6000 kg.

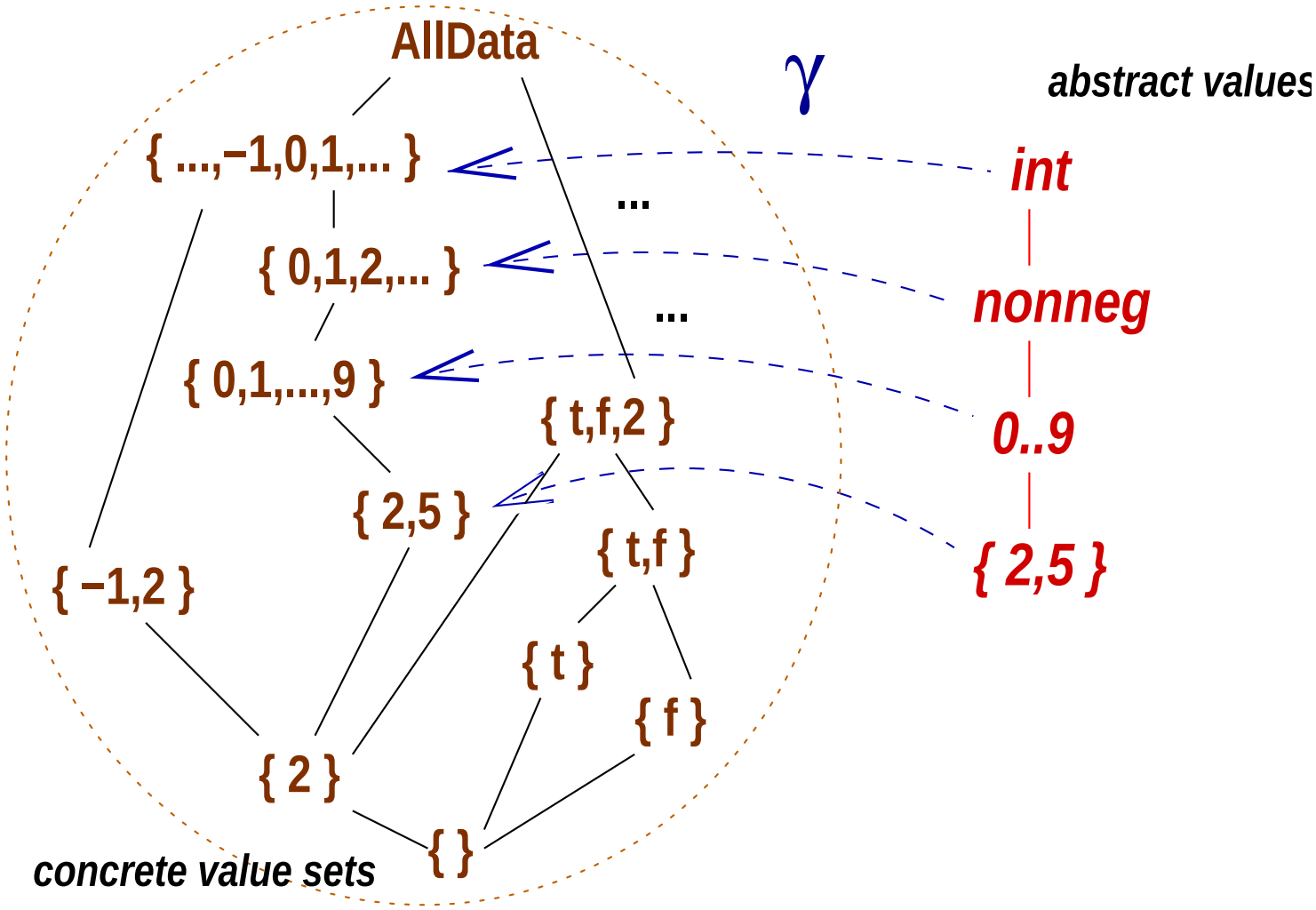
.

In computing, we use value abstractions



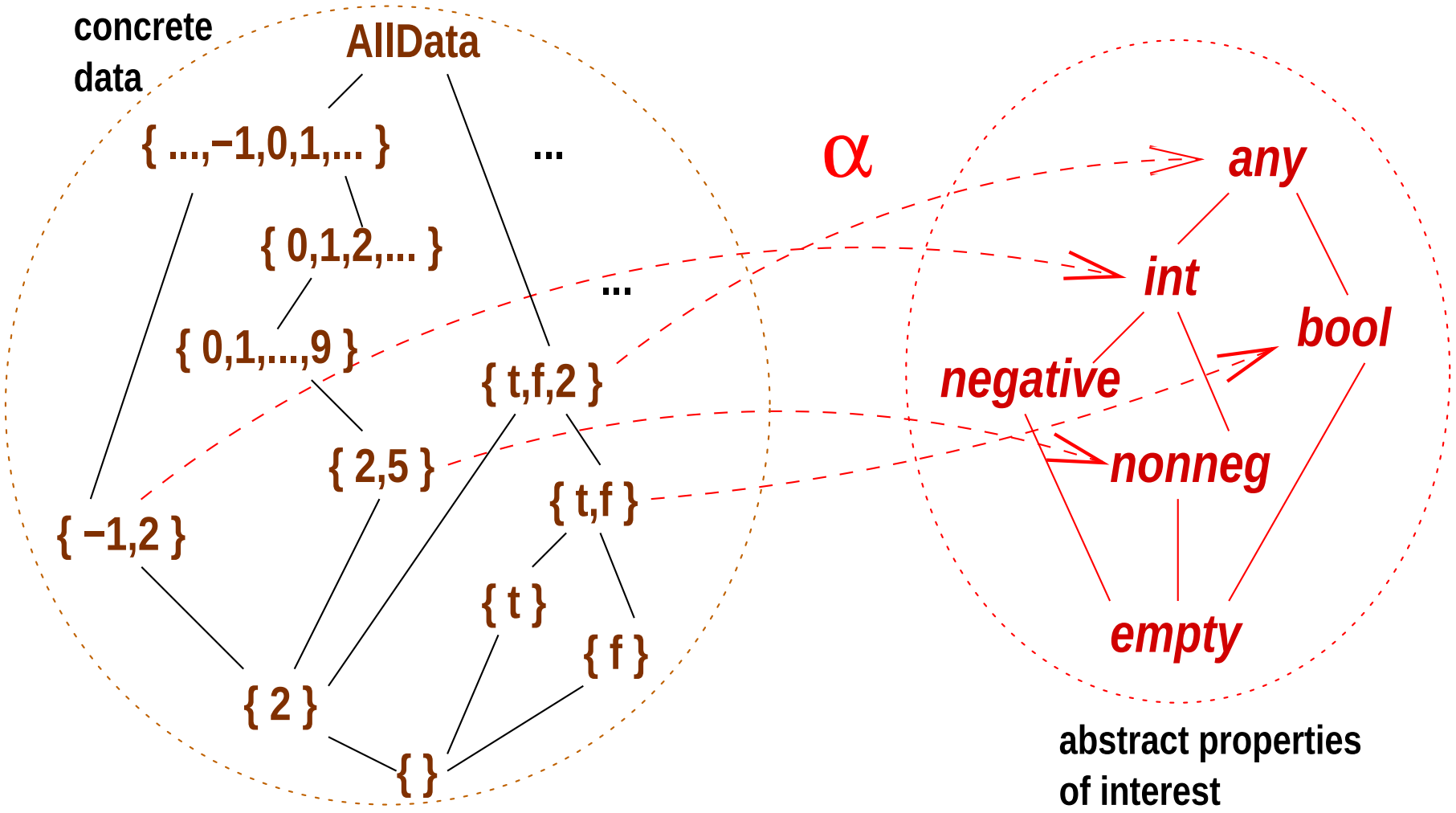
All the properties listed on the right are abstractions of 2; the upwards lines denote $\sqsubseteq$, a loss of precision.

Abstract values name sets of concrete values



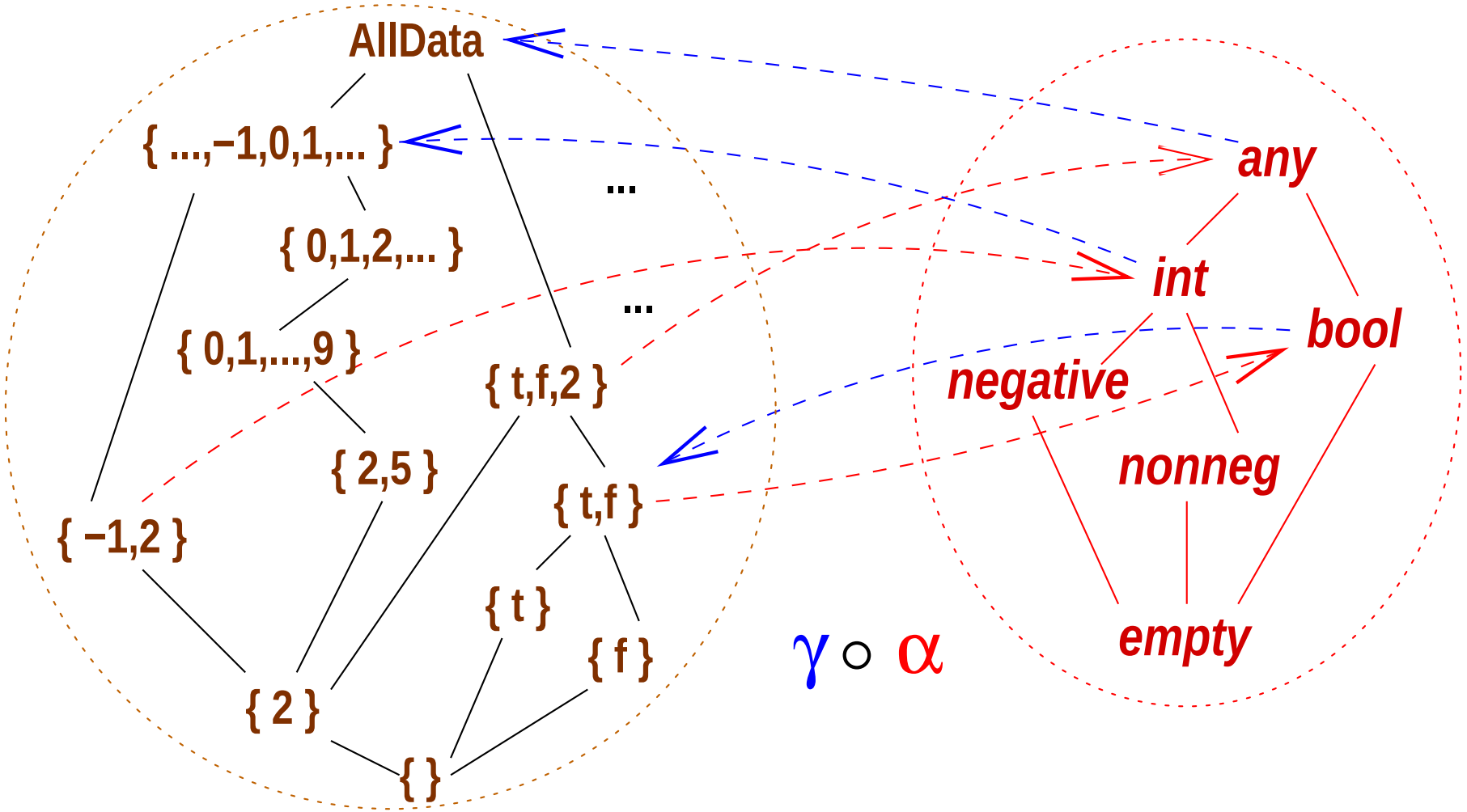
Function γ maps each abstract value to the set of concrete values it represents.

Sets of concrete values are abstracted imprecisely



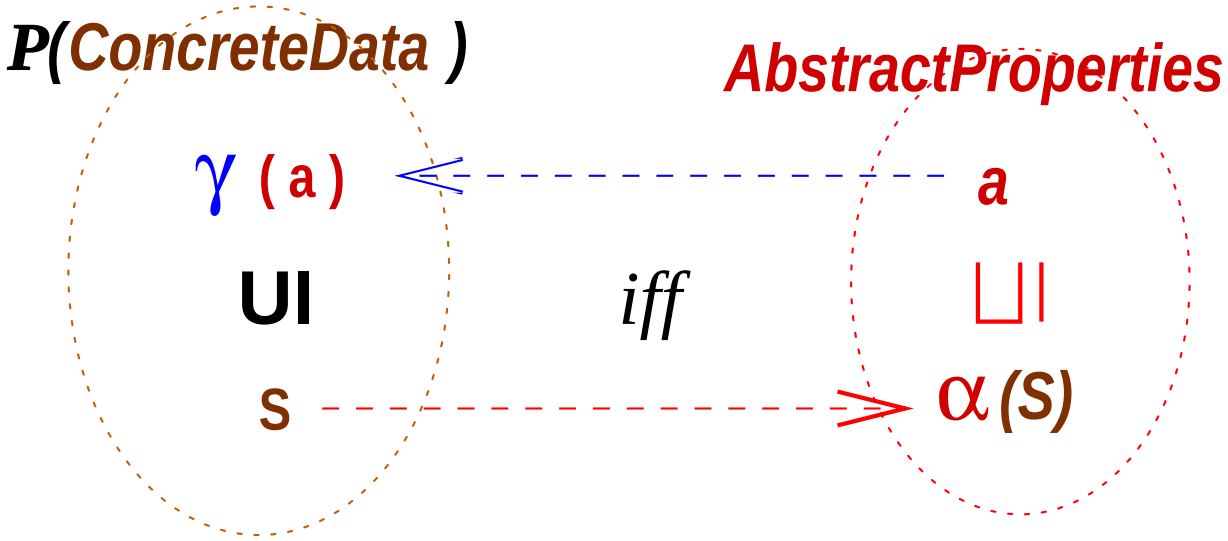
Function α maps each set to the abstract value that best describes it.

Abstraction followed by concretization demonstrates that α is sound but not exact



Nonetheless, the α given here is as precise as it possibly can be, given the abstract value domain and γ.

A Galois connection formalizes the situation



That is, for all $S \in \mathcal{P}(ConcreteData), a \in AbstractProperties$,

$$S \subseteq \gamma(a) \text{ iff } \alpha(S) \sqsubseteq a$$

When α and γ are monotone, this is equivalent to

$$S \subseteq \gamma \circ \alpha(S) \quad \text{and} \quad \alpha \circ \gamma(a) \sqsubseteq a$$

For practical reasons, the second inequality is usually restricted to $\alpha \circ \gamma(a) = a$, meaning that all abstract properties are "exact."

Perhaps the oldest application of abstract interpretation is to data-type checking

```
int x;
int[] a = new int[10];
 ...
a[0] = x + 2;  // Whatever x's run-time value might
 ...           //   be, we know it is an  int.
a[1] = (!x);   // Erroneous --- an  int  cannot be
               //   negated, nor can a  bool  be
               //   saved in an  int  cell.
```

But compilers struggle with imprecise abstractions

```
int x;
int[] a = new int[10];
 ...              // Because  x's  value is described
a[2 * x] = 3;     //  imprecisely, we cannot decide
                  //  whether  2 * x  falls in the
                  //  interval,  [0,9].
```

We might address array-indexing calculation by

1. making the abstraction more precise, e.g., declaring `x` with the abstract value ("data type") $[0, 9]$;
2. computing a "symbolic execution" of the program with the abstract values

These extensions underlie data-¤ow analyses and many sophisticated program analysis techniques.

A starting point: Trace-based operational semantics

```
p0 : while isEven(x) {
       p1 : x = x div 2;
     }
p2 : x = 4 * x;
p3 : exit
```

The operational semantics updates a program-point, storage-cell pair, pp, x, using these four transition rules:

$$p_0, 2n \longrightarrow p_1, 2n \qquad p_1, n \longrightarrow p_0, n/2$$

$$p_0, 2n+1 \longrightarrow p_2, 2n+1 \qquad p_2, n \longrightarrow p_3, 4n$$

A program's operational semantics is written as a trace:

$$p_0, 12 \longrightarrow p_1, 12 \longrightarrow p_0, 6 \longrightarrow p_1, 6 \longrightarrow p_0, 3 \longrightarrow p_2, 3 \longrightarrow p_3, 12$$

We can abstractly interpret, say, for polarity

```
p0 : while isEven(x) {
       p1 : x = x div 2;
     }
p2 : x = 4 * x;
p3 : exit
```

$p_0, even \longrightarrow p_1, even$

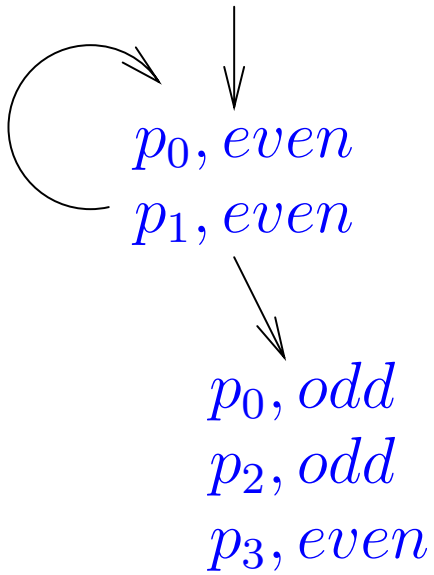
$p_0, odd \longrightarrow p_2, odd$

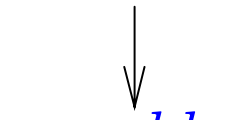
$p_1, even \longrightarrow p_0, even$

$p_1, even \longrightarrow p_0, odd$

$p_2, a \longrightarrow p_3, even$

Two trace trees cover the full range of inputs:





p_0, odd
p_2, odd
$p_3, even$

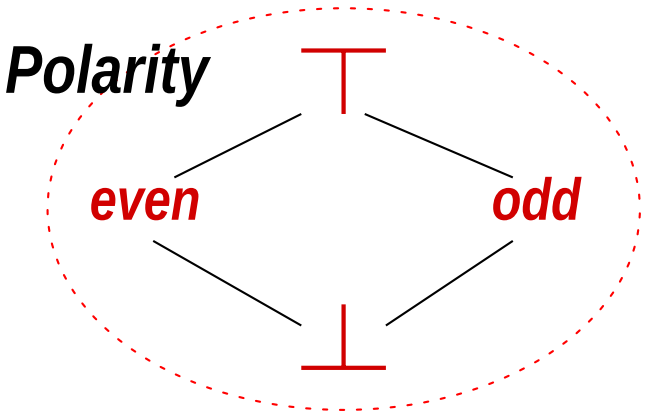
.

We conclude that

- if the program terminates, `x` is even-valued
- if the input is odd-valued, the loop will not be entered

Due to the loss of precision, we have not decided termination for even-valued inputs.

The underlying abstract interpretation



$$\gamma : Polarity \rightarrow \mathcal{P}(Int)$$

$$\gamma(even) = \{..., -2, 0, 2, ...\}$$

$$\gamma(odd) = \{..., -1, 1, 3, ...\}$$

$$\gamma(\top) = Int, \quad \gamma(\bot) = \{\ \}$$

$\alpha : \mathcal{P}(Int) \rightarrow Polarity$

$\alpha(S) = \sqcup\{\beta(v) | v \in S\}$, where $\beta(2n) = even$ and $\beta(2n+1) = odd$

The abstract transition rules are synthesized from the orginals:

$p_i, a \longrightarrow p_j, \alpha(v')$, if $v \in \gamma(a)$ and $p_i, v \longrightarrow p_j, v'$

This recipe ensures that every transition in the original, “concrete” semantics is simulated by one the abstract semantics.

To elaborate, remember that an abstract state, p_i, a, represents (abstracts) the set of concrete states,

$$\gamma_{State}(p_i, a) = \{p_i, c \mid c \in \gamma(a)\}$$

So, if some p_i, c in the above set can transit to p_j, c', then its abstraction must make a similar move:

$p_i, c \longrightarrow p_j, c'$ implies $p_i, a \longrightarrow p_j, a'$, where $p_j, c' \in \gamma_{State}(p_j, a')$.

Thus, the abstract semantics simulates all computations of the concrete semantics (and due to imprecision, produces more computations than are concretely possible).

Given a Galois connection, α, γ, we synthesize the most precise abstract semantics that simulates the concrete one as de£ned on the previous slide.

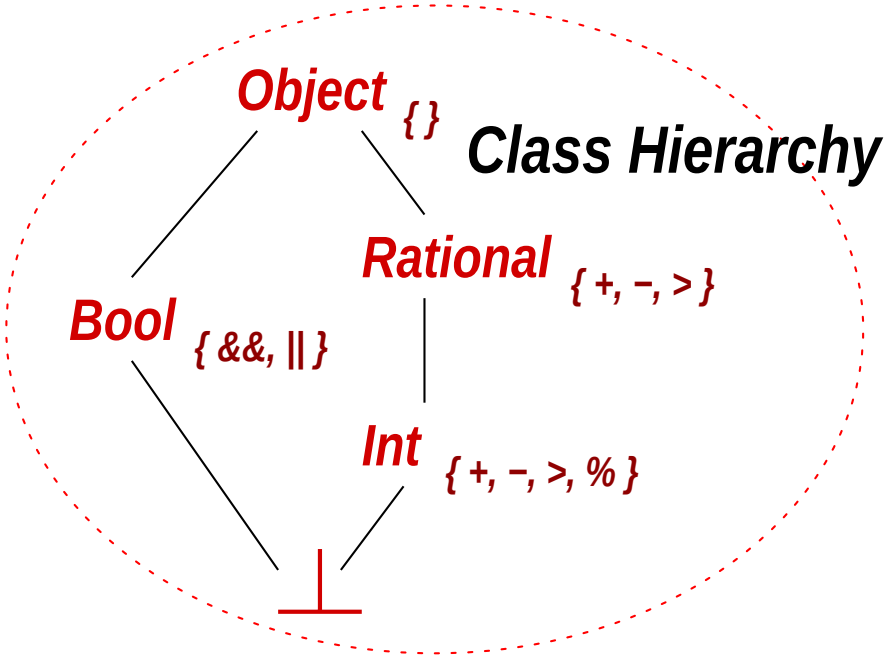
Abstract interpretation is ¤exible

We will apply abstract intepretation to

- data-type inference
- code improvement
- debugging
- assertion synthesis and program proving
- model-checking temporal logic formulas

Data-type compatibility inference

```
p0 : x = 4;
p1 : while ... {
       p2 : x = (x > 0)
     }
p3 : x = x % 2;
p4 : exit
```



$p_0, \tau \longrightarrow p_1, Int$

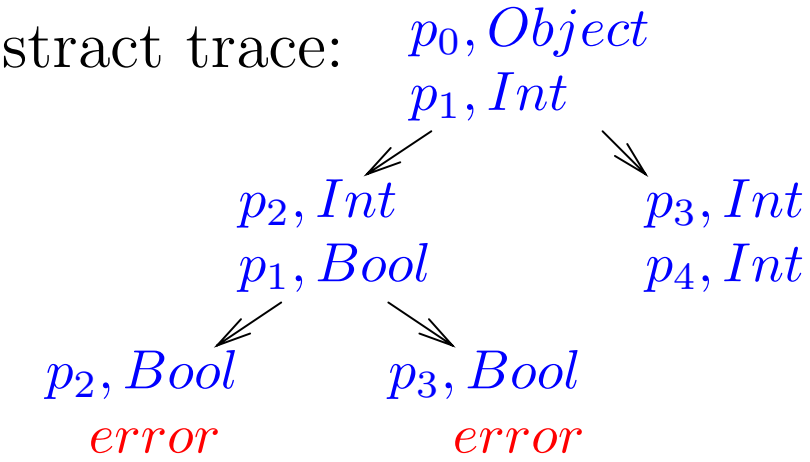
$p_1, \tau \longrightarrow p_2, \tau$

$p_1, \tau \longrightarrow p_3, \tau$

$p_2, \tau \longrightarrow p_1, Bool,$ if $\tau \sqsubseteq Rational$

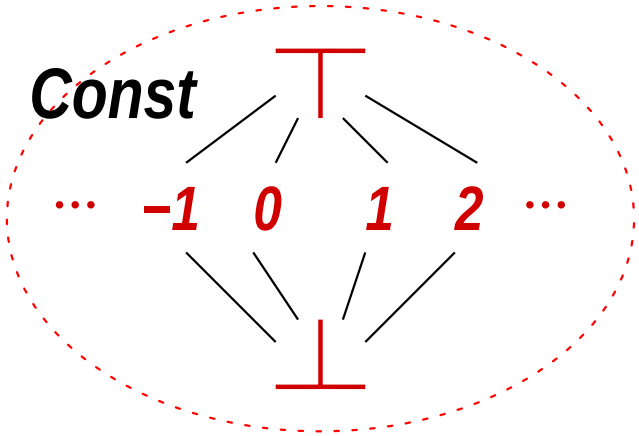
$p_3, Int \longrightarrow p_4, Int$

Abstract trace:



Constant propagation analysis

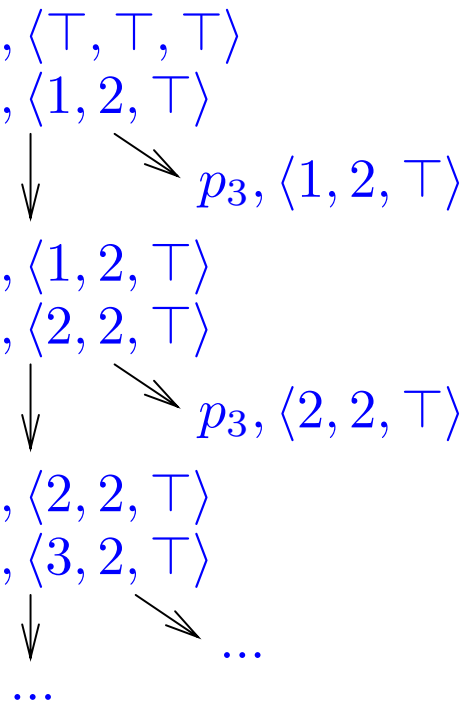
```
p0 : x = 1; y = 2;
p1 : while (x < y + z)
       p2 : x = x + 1;
     }
p3 : exit
```



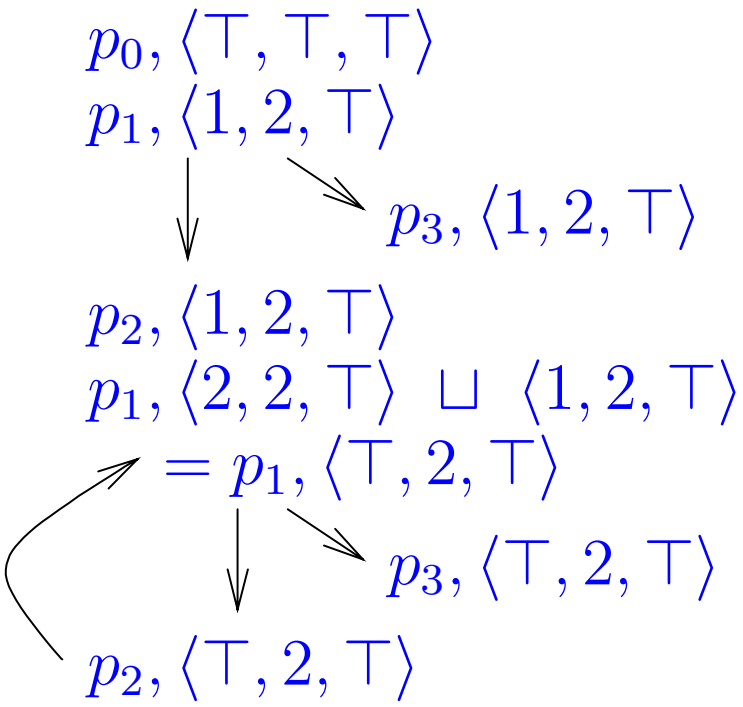
where $m + n$ is interpreted

$$k_1 + k_2 \longrightarrow sum(k_1, k_2),$$
$$\top \neq k_i \neq \bot, i \in 1..2$$
$$\top + k \longrightarrow \top$$
$$k + \top \longrightarrow \top$$

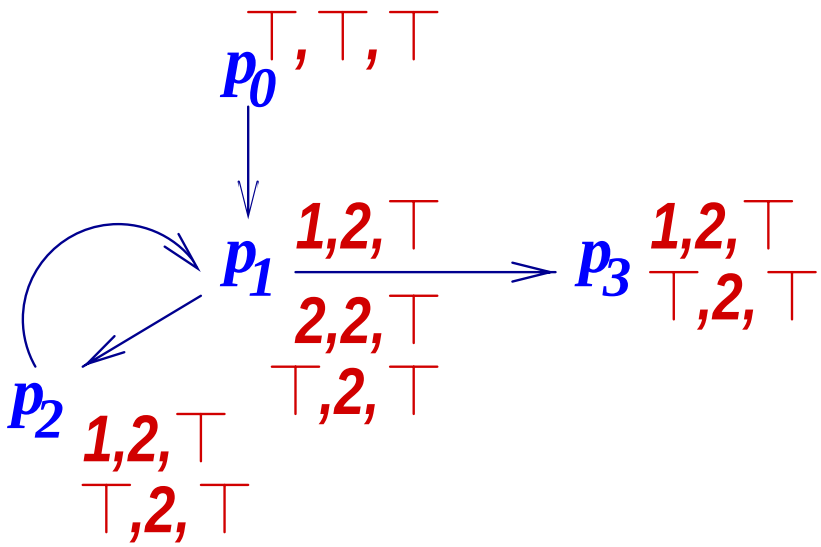
Abstract trace:



An acceleration is needed for £nite convergence



Drawn as a data−flow analysis:



The analysis tells us to replace y at p_1 by 2:

```
p0 : x = 1; y = 2;
p1 : while (x < 2 + z)
       p2 : x = x + 1;
     }
p3 : exit
```

Array bounds (pre)checking

Integer variables receive values from the interval domain,

$$I = \{[i, j] \mid i, j \in Int \cup \{-\infty, +\infty\}\}.$$

We de£ne $[a, b] \sqcup [a', b'] = [min(a, a'), max(b, b')]$.

```
int a = new int[10];
i = 0;
while (i < 10)  {
      ... a[i] ...
      i = i + 1;
}
```

$i = [0,0]$

p_1

$i = [0,0] \sqcap [-\infty, 9] = [0,0]$

$i = ([0,0] \sqcup [1,1]) \sqcap [-\infty, 9] = [0,1]$

...

p_2

$i = [1,1]$

$i = [1,1] \sqcup [2,2] = [1,2]$

...

This generates an in£nite sequence; we must accelerate with ∇:

p_1 : $i = [0,0] \sqcap [-\infty, 9] = [0,0]$

$i = ([0,0] \nabla [1,1]) \sqcap [-\infty, 9]$

$= [0, +\infty] \sqcap [-\infty, 9] = [0,9]$

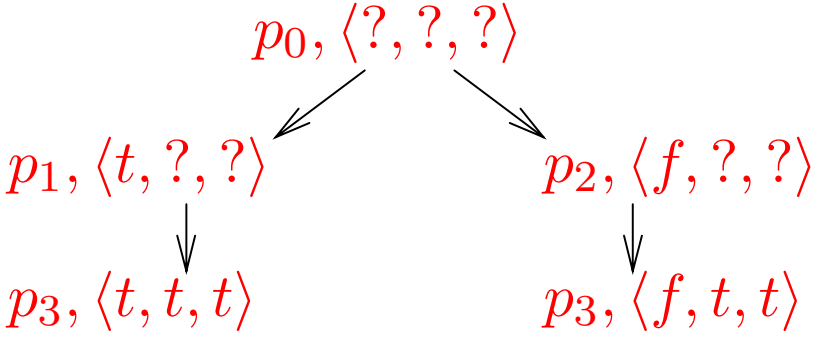
p_2 : $i = [1,1]$

$i = [1,1] \nabla [1,10] = [1, +\infty]$

Program veri£cation via predicate abstraction

We wish to prove that $\texttt{z} \geq \texttt{x} \wedge \texttt{z} \geq \texttt{y}$ at p_3:

```
p0 : if x < y
p1 :   then z = y
p2 :   else z = x
p3 : exit
```



We chose three predicates,

$$\phi_1 = \texttt{x} < \texttt{y}$$

$$\phi_2 = \texttt{z} \geq \texttt{x}$$

$$\phi_2 = \texttt{z} \geq \texttt{y}$$

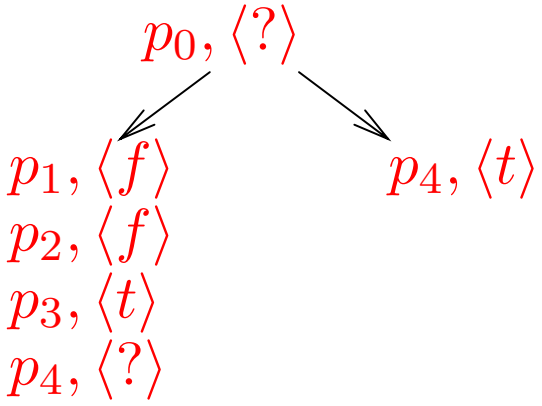
and computed their values at the program's points. The predicates' values come from the domain, $\{t, f, ?\}$. (Read $?$ as $t \vee f$.)

At all occurrences of p_3 in the abstract trace, $\phi_2 \wedge \phi_3$ holds.

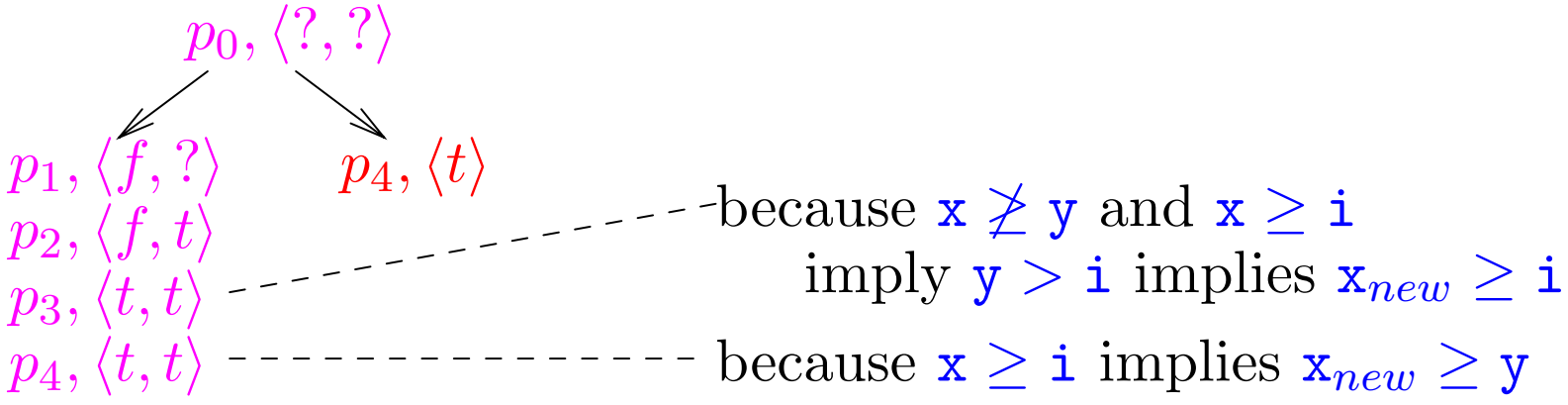
When a goal is undecided, re£nement is necessary

Prove $\phi_0 \equiv \mathtt{x} \geq \mathtt{y}$ at p_4:

```
p0 : if !(x >= y)
p1 : then { i = x;
p2 :        x = y;
p3 :        y = i;
p4 : }
```



To decide the goal, we must re£ne the state by adding a needed auxiliary predicate: $wp(\mathtt{y} = \mathtt{i}, \mathtt{x} \geq \mathtt{y}) = (\mathtt{x} \geq \mathtt{i}) \equiv \phi_1$.



But incremental predicate re£nement cannot synthesize many interesting loop invariants. For this example:

```
p0 : i = n; x = 0;
p1 : while  i != 0  {
   p2 : x = x + 1;  i = i - 1;
     }
p3 : goal: x = n
```

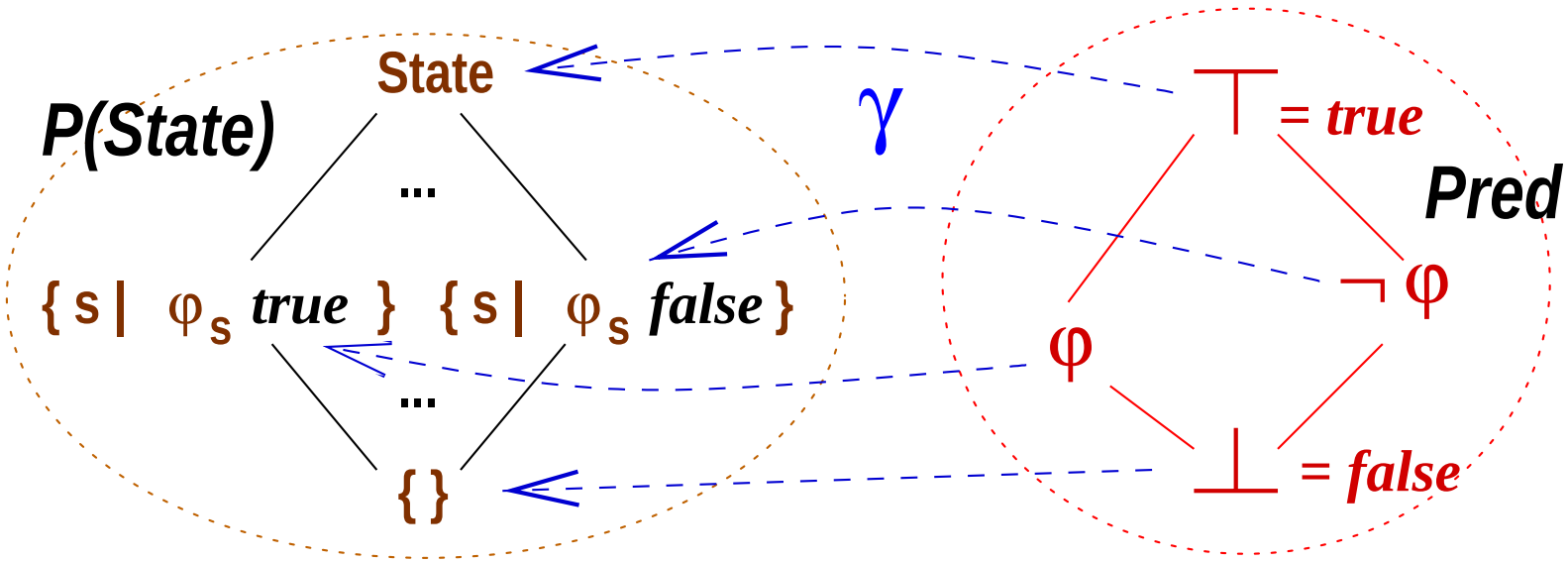
We £nd that the initial predicate set, $P_0 \equiv \{\mathtt{i} = 0, \mathtt{x} = n\}$, does not validate the loop body.

The £rst re£nement suggests we add $P_1 \equiv \{\mathtt{i} = 1, \mathtt{x} = n - 1\}$ to the program state, but this fails to validate a loop that iterates more than once.

Re£nement stage j adds predicates $P_j \equiv \{\mathtt{i} = j, \mathtt{x} = n - j\}$; the re£nement process continues forever!

The loop invariant is $\mathtt{x} = \mathtt{n} - \mathtt{i}$:-)

Predicates live in an abstract domain with a negation operation



The domain is a boolean lattice; a stronger condition is boolean algebra (where the distributive laws hold).

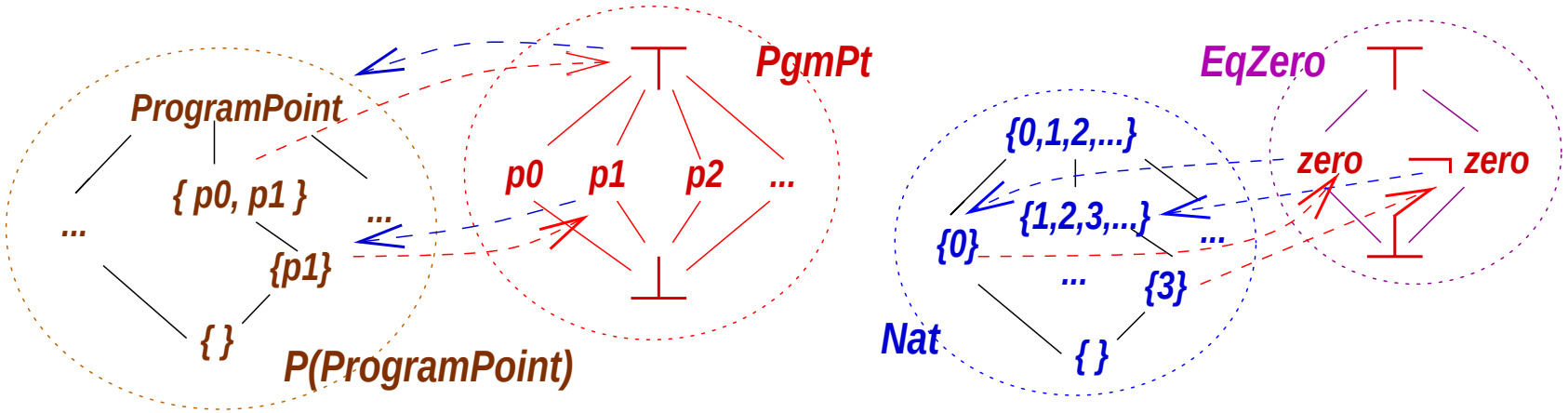
If we check for correctness criterion, $S_c \subseteq \mathcal{P}(State)$, then $\alpha(S_c)$ must be exact — $\gamma \circ \alpha(S_c) = S_c$ — to use $\phi_c = \alpha(S_c)$ in the abstract analysis. (Otherwise, we might incorrectly infer, from p', ϕ_c, that all such p', s', where $s' \in \gamma(\phi_c)$, are "correct.")

Every abstract domain de£nes a “logic”

For abstract domain A, $a \in A$ is a “property/predicate,” and $\gamma(a)$ de£nes those concrete states that make a “true”:

$$s \; has \; a \text{ , written } \; s \models_A a, \text{ iff } \; \alpha\{s\} \sqsubseteq a$$

Example: for concrete states, $ProgramPoint \times Nat$, and its abstraction, $PgmPt \times EqZero$:



We have these facts: $p_1 \models_{PgmPt} p_1$ and $3 \models_{EqZero} \neg zero$

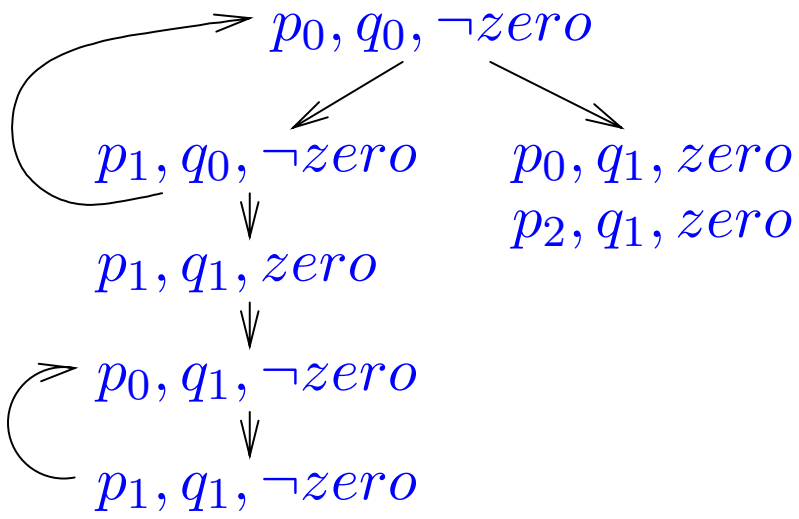
therefore, $p_1, 3 \models_{PP \times EqZero} p_1, \neg zero$

Abstract traces are examples of model checks

p_0 : `while x > 0 {`
p_1 : *use resource*
`x = x + 1; }`
p_2 : *sleep*

`||`

q_0 : `x = 0;`
q_1 : *use resource*



Say that p_1, q_1 is an error state; is there a path from the start to it:

$$p_0, q_0, k \models_{LTL(PgmPt \times EqZero)} F(p_1, q_1, \top) \text{ ?}$$

Say that p_2 is a rest state; for all states reached from the start, can we progress to it:

$$p_0, q_0, k \models_{LTL(PgmPt \times EqZero)} GF(p_2, \top, \top) \text{ ?}$$

The logical operators, G and F, describe reachability properties in the temporal logic, LTL.

A state, s_0, names the set of traces that begin with it. An LTL property, ϕ, describes a pattern of states in a trace.

$s_0 \models \phi$ means that all traces, $s_0 \rightarrow s_1 \rightarrow \cdots$, contain pattern ϕ.

MiniLTL: $\phi ::= a \mid G\phi \mid F\phi$ Semantics: $[\![\phi]\!] \subseteq \mathcal{P}(Trace)$

$$[\![a]\!] = \{\pi \mid \pi_0 \models_A a\}$$
$$[\![G\phi]\!] = \{\pi \mid \forall i \geq 0, \pi \downarrow i \in [\![\phi]\!]\}$$
$$[\![F\phi]\!] = \{\pi \mid \exists i \geq 0, \pi \downarrow i \in [\![\phi]\!]\}$$

where, for $\pi = s_0 \rightarrow s_1 \rightarrow \cdots$, let $\pi_0 = s_0$ and $\pi \downarrow i = s_i \rightarrow s_{i+1} \rightarrow \cdots$.

There is a Galois connection, $(\mathcal{P}(Trace), \subseteq) \leftrightarrow (\mathcal{P}(\text{MiniLTL}), \supseteq)$, where $\sqcup = \cap = \wedge$ in $\mathcal{P}(\text{MiniLTL})$:

$$\gamma(P) = \bigcap\{[\![\phi]\!] \mid \phi \in P\}$$
$$\beta(\pi) = \{\phi \mid \pi \in \gamma\{\phi\}\}$$
$$\alpha(S) = \bigcap\{\beta(\pi) \mid \pi \in S\}$$

But this is just the beginning of a long story about the relationship of abstract interpretation to temporal-logic model checking!

Every concrete value is the conjunction of its abstractions (its "abstract-interpretation DNA")

$$= elephant_{species} \wedge brown_{color} \wedge heavy_{weight}$$
$$\wedge\, 4000..6000kg_{weight} \wedge \cdots$$

There is even a pattern of Galois connection for this:

$$\gamma : AllPossibleProperties \rightarrow \mathcal{P}(RealWorldObjects)$$
$$\gamma(p) = \{c \in RealWorldObjects \mid c\ has\ property\ p\}$$

$$\beta : RealWorldObjects \rightarrow AllPossibleProperties$$
$$\beta(c) = \sqcap\{p \in AllPossibleProperties \mid c \in \gamma(p)\}$$

$$\alpha : \mathcal{P}(RealWorldObjects) \rightarrow AllPossibleProperties$$
$$\alpha(S) = \sqcup\{\beta(s) \mid s \in S\}$$

Some references

- The papers of Patrick and Radhia Cousot (www.di.ens.fr/~cousot), but especially
  1. Abstract interpretation: a uni£ed lattice model for static analysis of programs by construction or approximation of £xpoints. ACM POPL 1977.
  2. Systematic design of program analysis frameworks. ACM POPL, 1979.
  3. Temporal Abstract Interpretation. ACM POPL, 2000.
- Neil Jones and Flemming Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In Handbook of Logic in Computer Science, Vol. 4, Oxford University Press, 1994.
- A few of my papers, found at www.cis.ksu.edu/~schmidt/papers:
  1. Trace-Based Abstract Interpretation of Operational Semantics. J. Lisp and Symbolic Computation 10-3 (1998).
  2. Data-¤ow analysis is model checking of abstract interpretations. ACM POPL 1998.
  3. From Trace Sets to Modal-Transition Systems by Stepwise Abstract

Interpretation. Workshop on Structure Preserving Relations, Amagasaaki, Japan, 2001.

4. Structure-preserving binary relations for program abstraction. In The Essence of Computation: Complexity, Analysis, Transformation. Springer LNCS 2566, 2002.