Introduction to Abstraction and Static Analysis

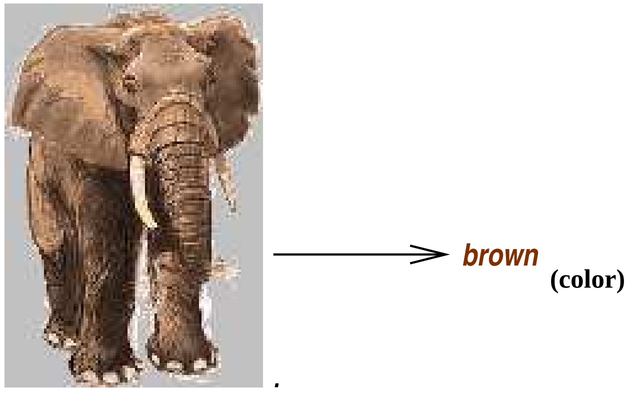
David Schmidt
Kansas State University

www.cis.ksu.edu/~schmidt

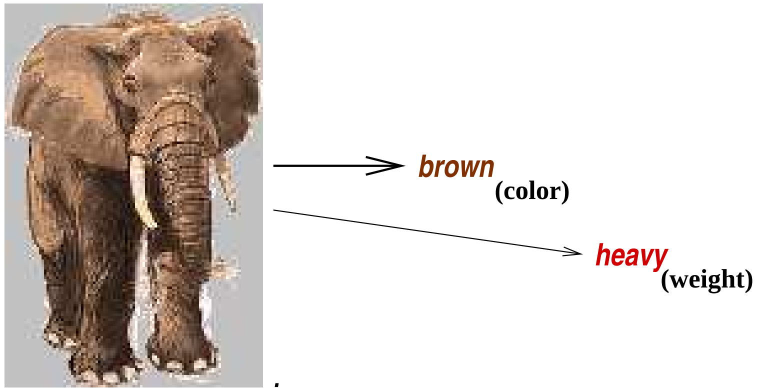
Outline

1. What is abstraction?
2. Abstraction and concretization:
   Galois-connection-based abstract interpretation
3. Examples of static analyses
4. Logics and static analysis

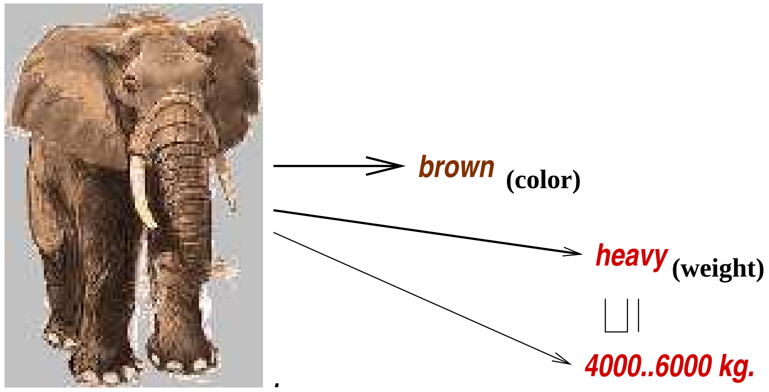
An *abstraction* is a property from some domain



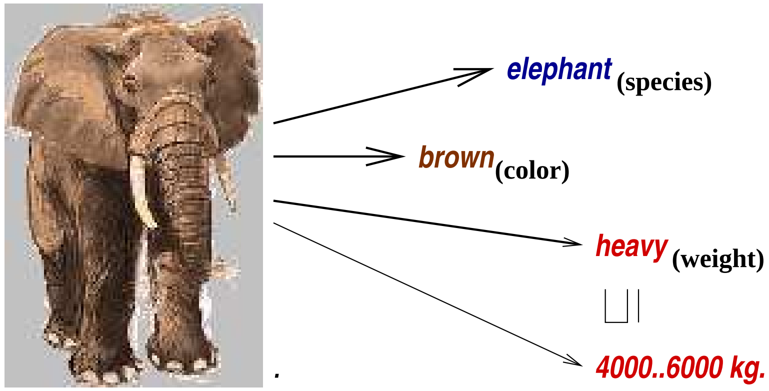
An *abstraction* is a property (cont.)



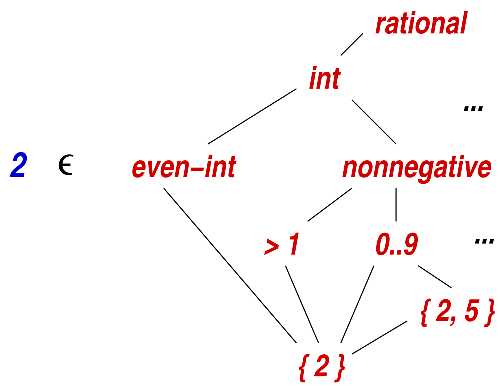
An *abstraction* is a property (cont.)



An *abstraction* is a property (concl.)

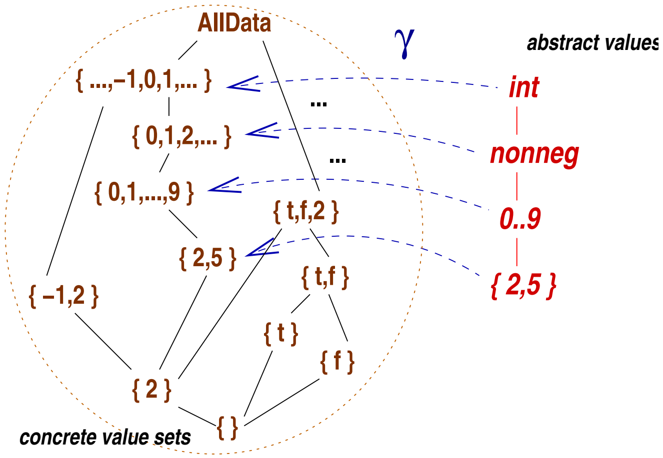


Value abstractions are classic to computing



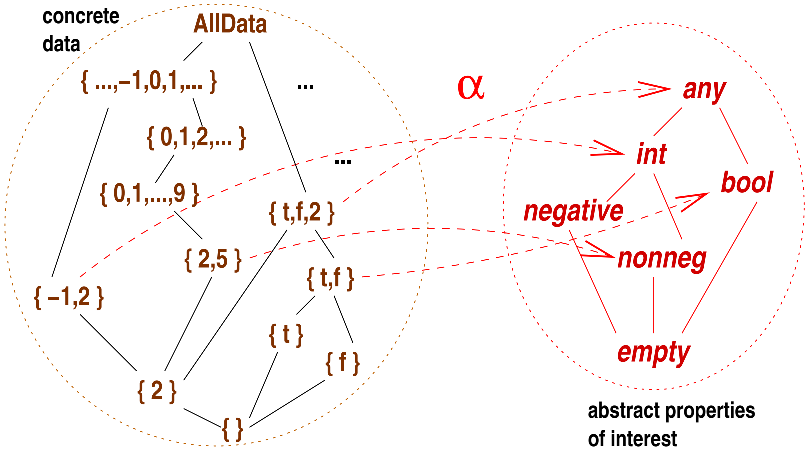
All the properties listed on the right are abstractions of 2; the upwards lines denote $\sqsubseteq$, a loss of precision.

Abstract values name sets of concrete values



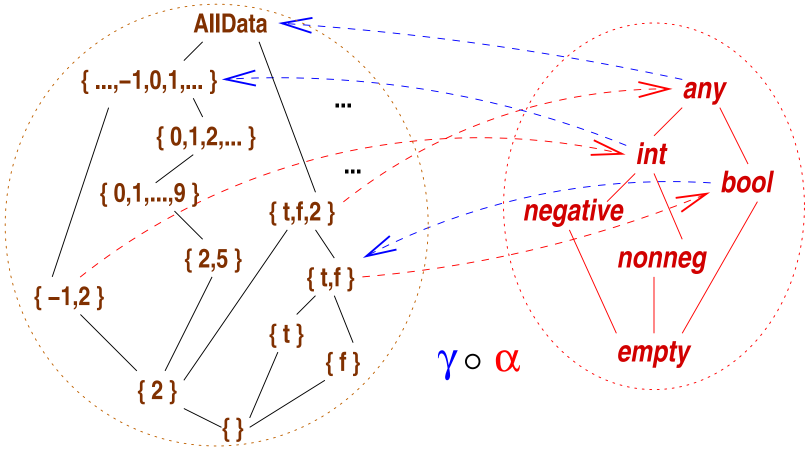
Function γ maps each abstract value to the set of concrete values it represents.

Sets of concrete values are abstracted imprecisely



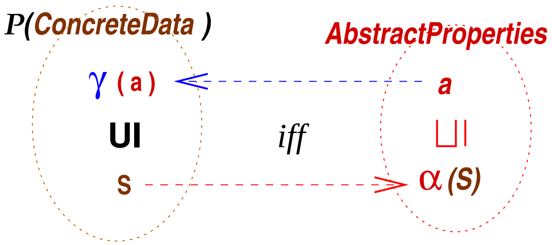
Function α maps each set to the abstract value that best describes it.

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 $S \in \mathcal{P}(\mathrm{ConcreteData})$, $a \in \mathrm{AbstractProperties}$,

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

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 employ 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-flow 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:

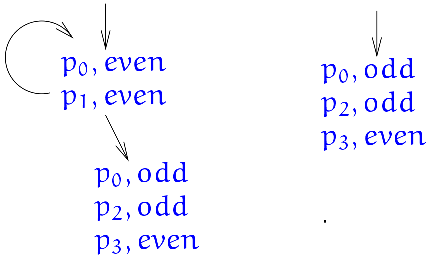
$$p_0, 12 \longrightarrow p_1, 12 \longrightarrow p_0, 6 \longrightarrow p_1, 6 \longrightarrow p_0, 3 \longrightarrow p_2, 3 \longrightarrow p_3, 12$$

We can abstractly interpret, say, for parity

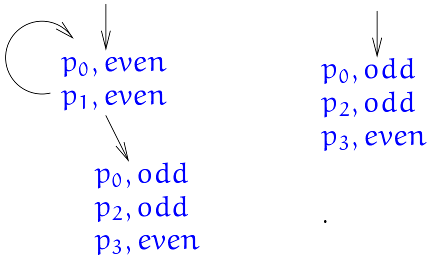
```
p0 : while isEven(x) {
       p1 : x = x div 2;
     }
p2 : x = 4 * x;
p3 : exit
```

$$p_0, even \longrightarrow p_1, even$$
$$p_0, odd \longrightarrow p_2, odd$$
$$p_1, even \longrightarrow p_0, even$$
$$p_1, even \longrightarrow p_0, odd$$
$$p_2, a \longrightarrow p_3, even$$

Two trace trees cover the full range of inputs:



The interpretation of the program's semantics with the abstract values is an *abstract interpretation*:

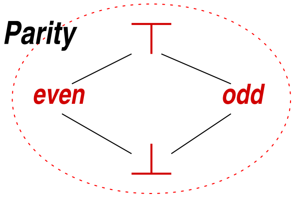


We conclude that

- if the program terminates, `x` is even-valued
- if the input is odd-valued, the loop body, p_1, will not be entered

Due to the loss of precision, we can not decide termination for almost all the even-valued inputs. (Indeed, only 0 causes nontermination.)

The underlying abstract-interpretation semantics



$\gamma : \mathrm{Parity} \to \mathcal{P}(\mathrm{Int})$

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

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

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

$\alpha : \mathcal{P}(\mathrm{Int}) \to \mathrm{Parity}$

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

The abstract transition rules are synthesized from the orginals:

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

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

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

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

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

Abstract interpretation underlies most *static analyses*

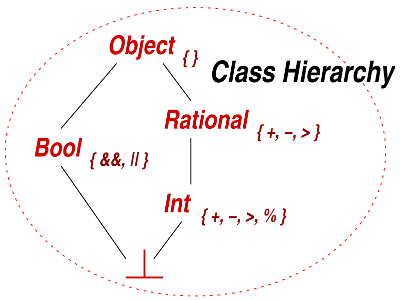
A *static analysis* of a program is a *sound, finite, and approximate* calculation of the program’s executions. The trace trees we just generated for the loop program is an example of a static analysis.

We will survey static analyses for

- 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
```



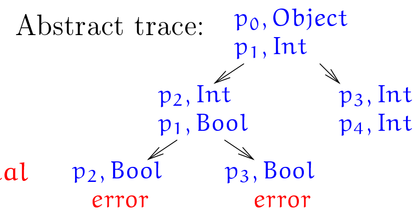
$p_0, \tau \longrightarrow p_1, \mathrm{Int}$

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

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

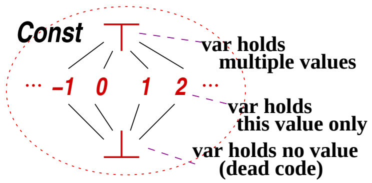
$p_2, \tau \longrightarrow p_1, \mathrm{Bool}, \text{ if } \tau \sqsubseteq \mathrm{Rational}$

$p_3, \mathrm{Int} \longrightarrow p_4, \mathrm{Int}$



Constant propagation analysis

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



where $m + n$ is interpreted

$k_1 + k_2 \longrightarrow sum(k_1, k_2)$,

$\top \neq k_i \neq \perp, i \in 1..2$

$\top + k \longrightarrow \top$

$k + \top \longrightarrow \top$

Let $\langle u, v, w\rangle$ abbreviate

$\langle x : u, y : v, z : w\rangle$

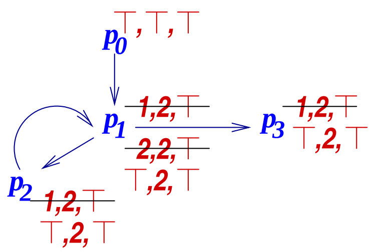
Abstract trace:

$p_0, \langle \top, \top, \top\rangle$
$p_1, \langle 1, 2, \top\rangle$ → $p_3, \langle 1, 2, \top\rangle$
$p_2, \langle 1, 2, \top\rangle$
$p_1, \langle 2, 2, \top\rangle$ → $p_3, \langle 2, 2, \top\rangle$
$p_2, \langle 2, 2, \top\rangle$
$p_1, \langle 3, 2, \top\rangle$ → ...
...

An *acceleration* is needed for finite convergence

$p_0, \langle \top, \top, \top\rangle$
$p_1, \langle 1, 2, \top\rangle$ → $p_3, \langle 1, 2, \top\rangle$
$p_2, \langle 1, 2, \top\rangle$
$p_1, \langle 2, 2, \top\rangle \sqcup \langle 1, 2, \top\rangle$
$= p_1, \langle \top, 2, \top\rangle$ → $p_3, \langle \top, 2, \top\rangle$
$p_2, \langle \top, 2, \top\rangle$

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 < y + z)  {
       p2 : x = x + 1;
     }
p3 : exit
```

2

Array bounds (pre)checking uses intervals

Integer variables receive values from the *interval domain*,

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

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

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

i = [0,0]

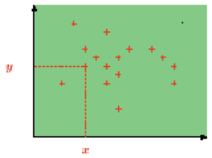
p_1: *i = [0,0] ⊓ [− ∞,9] = [0,0]*
i = [0,0] ⊔ [1,1] ⊓ [− ∞,9] = [0,1]
...

p_2: *i = [1,1]*
i = [1,1] ⊔ [2,2] = [1,2]
...

At convergence, i's ranges are
at p_1 : $[0..9]$
at p_2 : $[1..10]$
at loop exit : $[1..10] \sqcap [10, +\infty] = [10, 10]$

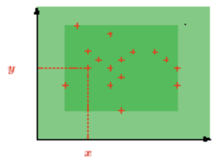
Examples of relations between variables' values

These Figures are from *Abstract Interpretation: Achievements and Perspectives* by Patrick Cousot, Proc. SSGRR 2000.



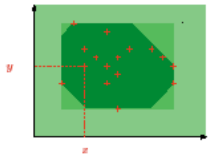
$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$

Fig. 2
SIGNS



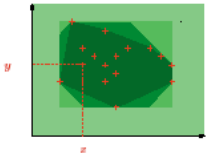
$\begin{cases} x \in [3,\ 27] \\ y \in [4,\ 32] \end{cases}$

Fig. 3
INTERVALS



$\begin{cases} 3 \leq x \leq 27 \\ x + y \leq 88 \\ 4 \leq y \leq 32 \\ x - y \leq 61 \end{cases}$

Fig. 4
OCTAGONS



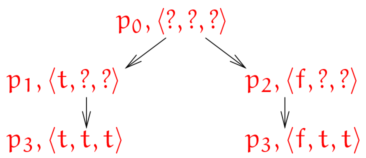
$\begin{cases} 7x + 31y \leq 325 \\ 21x + 7y \geq 0 \end{cases}$

Fig. 5
POLYHEDRA

Program verification via *predicate abstraction*

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

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



We choose three predicates,

$$\phi_1 = x < y$$
$$\phi_2 = z \geq x$$
$$\phi_2 = z \geq y$$

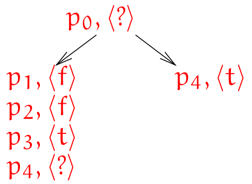
and compute their values at the program's points. The predicates' values come from the domain, $\{t, f, ?\}$. (Read $?$ as $t \vee f$.)

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

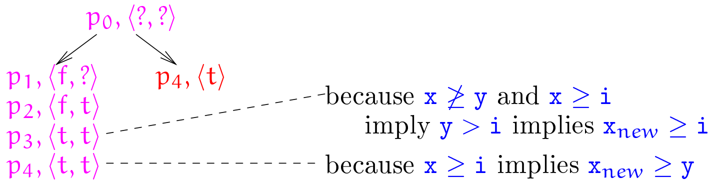
When a goal is undecided, *refinement* is necessary

Prove $\phi_0 \equiv x \geq y$ at p_4:

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



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



But incremental predicate refinement 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 find that the initial predicate set, $P_0 \equiv \{i = 0, x = n\}$, does not validate the loop body.

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

Refinement stage j adds predicates $P_j \equiv \{i = j, x = n - j\}$; the refinement process continues forever!

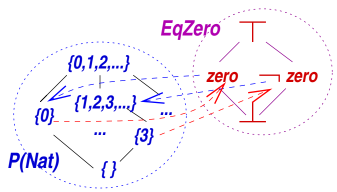
The loop invariant is $x = n - i$:-)

An abstract domain defines a "logic"

For abstract domain A, $a \in A$ is a "property/predicate," and $\gamma(a) \subseteq C$ defines a subset of concrete states that make a "true." For $s \in C$,

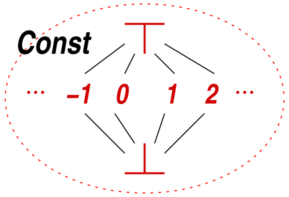
$$s \textit{ has } a \text{ , written } s \models_A a, \text{ iff } s \in \gamma(a) \text{ iff } \alpha\{s\} \sqsubseteq a$$

Example: We might abstract Nat by $EqZero$:



We have, for example, that $3 \models \neg zero$; we also have that $3 \models \top$; and we have that $3 \models \neg zero \sqcap \top$.

In one sense, *every* analysis based on abstract interpretation is a "predicate abstraction." But the "logic" is weak — it supports conjunction ($\sqcap$) but not necessarily disjunction ($\sqcup$).



For **Const**, we have that

$2 \models_{Const} 2 \sqcap \top$ ***iff*** $2 \models_{Const} 2$ ***and*** $2 \models_{Const} \top$.

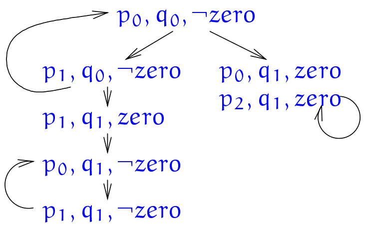
In general, $n \models_{Const} a \sqcap a'$ iff $n \models_{Const} a$ and $n \models_{Const} a'$

But **Const** does not support disjunction: $2 \models_{Const} \top$, and $\top = 2 \sqcup 3 = 3 \sqcup 4 = 2 \sqcup 3 \sqcup 4$, etc.

Hence $2 \models_{Const} 3 \sqcup 4$, but this does *not* imply that $2 \models_{Const} 3$ or $2 \models_{Const} 4$!

Abstract traces can be *model checked*

```
p0 : while x > 0 {        ||  q0 : x = 0;
  p1 : use resource            q1 : use
       x = x + 1;   }               resource
p2 : sleep forever                  forever
```



Starting from p_0, q_0, k, for $k > 0$, will every execution "Generally/Globally" avoid resource misuse ?

$$p_0, q_0, k \models G\, \neg(p_1 \wedge q_1) \text{ ?}$$

Will every execution reach a Future state where x is permanently (Generally/Globally) zero?

$$p_0, q_0, k \models FG\; zero \text{ ?}$$

The logical operators, F and G, 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, $s_0 \to s_1 \to \cdots$, 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 $\pi = s_0 \to s_1 \to \cdots$, let $\pi_0 = s_0$ and $\pi \downarrow i = s_i \to s_{i+1} \to \cdots$.

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

$\gamma(P) = \bigcap\{[\![\phi]\!] \mid \phi \in P\}$ – the traces that have all the properties in P

$\alpha(S) = \{\phi \mid S \subseteq [\![\phi]\!]\}$ – properties held by all traces 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")*

$$= \mathsf{elephant}_{species} \wedge \mathsf{brown}_{color} \wedge \mathsf{heavy}_{weight} \wedge 4000..6000\mathsf{kg}_{weight} \wedge \cdots$$

There is even a pattern of Galois connection for this:

$\gamma : \mathsf{AllPossibleProperties} \to \mathcal{P}(\mathsf{RealWorldObjects})$

$\quad \gamma(p) = \{c \in \mathsf{RealWorldObjects} \mid c \textit{ has property } p\}$

$\beta : \mathsf{RealWorldObjects} \to \mathsf{AllPossibleProperties}$

$\quad \beta(c) = \sqcap\{p \in \mathsf{AllPossibleProperties} \mid c \in \gamma(p)\}$

$\alpha : \mathcal{P}(\mathsf{RealWorldObjects}) \to \mathsf{AllPossibleProperties}$

$\quad \alpha(S) = \sqcup\{\beta(s) \mid s \in S\}$

References

- The papers of Patrick and Radhia Cousot (www.di.ens.fr/~cousot), including
  1. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. ACM POPL 1977.
  2. Systematic design of program analysis frameworks. ACM POPL, 1979.
  3. Abstract interpretation: achievements and perspectives. Proc. SSGRR 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.
- Hanne Nielson, Flemming Nielson, and Chris Hankin. Principles of Program Analysis. Springer 1999.
- 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-flow analysis is model checking of abstract interpretations. ACM POPL 1998.