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

David Schmidt
Kansas State University
An abstraction is a property from some domain
An abstraction is a property (cont.)

- **brown** (color)
- **heavy** (weight)
An abstraction is a property (cont.)

brown (color)

heavy (weight)

4000..6000 kg.
An abstraction is a property (concl.)

- elephant (species)
- brown (color)
- heavy (weight)
- 4000..6000 kg.
In computing, we use value abstractions

All the properties listed on the right are abstractions of 2; the upwards lines denote \( \subseteq \), a loss of precision.
Abstract values name sets of concrete values

Function $\gamma$ maps each abstract value to the set of concrete values it represents.
Sets of concrete values are abstracted imprecisely.

Function $\alpha$ maps each set to the abstract value that best describes it.
Abstraction followed by concretization demonstrates that $\alpha$ is sound but not exact.

Nonetheless, the $\alpha$ given here is as precise as it possibly can be, given the abstract value domain and $\gamma$. 
A Galois connection formalizes the situation

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

$$S \subseteq \gamma(a) \iff \alpha(S) \subseteq a$$

When $\alpha$ and $\gamma$ are monotone, this is equivalent to

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

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

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

```java
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

\[ p_0 : \text{while isEven}(x) \{ \]
\[ \quad p_1 : x = x \div 2; \]
\[ \}
\[ p_2 : x = 4 * x; \]
\[ p_3 : \text{exit} \]

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

\[ p_0, 2n \rightarrow p_1, 2n \]
\[ p_0, 2n + 1 \rightarrow p_2, 2n + 1 \]
\[ p_1, n \rightarrow p_0, n/2 \]
\[ p_2, n \rightarrow p_3, 4n \]

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

\[ p_0, 12 \rightarrow p_1, 12 \rightarrow p_0, 6 \rightarrow p_1, 6 \rightarrow p_0, 3 \rightarrow p_2, 3 \rightarrow p_3, 12 \]
We can abstractly interpret, say, for polarity

\[ p_0 : \text{while isEven}(x) \{ \]
\[ \quad p_1 : x = x \div 2; \]
\[ \} \]
\[ p_2 : x = 4 \times x; \]
\[ p_3 : \text{exit} \]

Two trace trees cover the full range of inputs:

\[ p_0, \text{even} \rightarrow p_1, \text{even} \]
\[ p_0, \text{odd} \rightarrow p_2, \text{odd} \]
\[ p_1, \text{even} \rightarrow p_0, \text{even} \]
\[ p_1, \text{even} \rightarrow p_0, \text{odd} \]
\[ p_2, a \rightarrow p_3, \text{even} \]
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

\[ \gamma : \text{Polarity} \rightarrow \mathcal{P}(\text{Int}) \]

- \( \gamma(\text{even}) = \{\ldots, -2, 0, 2, \ldots\} \)
- \( \gamma(\text{odd}) = \{\ldots, -1, 1, 3, \ldots\} \)
- \( \gamma(\top) = \text{Int}, \quad \gamma(\bot) = \{\} \)

\[ \alpha : \mathcal{P}(\text{Int}) \rightarrow \text{Polarity} \]

\[ \alpha(S) = \bigsqcup \{ \beta(v) | v \in S \}, \text{ where } \beta(2n) = \text{even and } \beta(2n + 1) = \text{odd} \]

The abstract transition rules are synthesized from the orginals:

\[ p_i, a \rightarrow p_j, \alpha(v') \text{, if } v \in \gamma(a) \text{ and } p_i, v \rightarrow p_j, v' \]

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,

\[
\gamma_{\text{State}}(p_i, a) = \{ p_i, c \mid c \in \gamma(a) \}
\]

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

\[ p_i, c \rightarrow p_j, c' \text{ implies } p_i, a \rightarrow p_j, a' \text{, where } p_j, c' \in \gamma_{\text{State}}(p_j, a'). \]

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, \( \alpha, \gamma \), we synthesize the most precise abstract semantics that simulates the concrete one as described on the previous slide.
Abstract interpretation is flexible

We will apply abstract interpretation to

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

\( p_0 : \ x = 4; \)
\( p_1 : \ while \ ... \ { \)
\( \quad p_2 : \ x = (x > 0) \)
\( \} \)
\( p_3 : \ x = x \% 2; \)
\( p_4 : \ exit \)

Class Hierarchy

Object \{\} 
Rational \{\ +, -, >\} 
Bool \{\ &&, ||\} 
Int \{\ +, -, >, %\}

Abstract trace:
\( p_0, Object \)
\( p_1, Int \)
\( p_1, Int \)
\( p_2, Int \)
\( p_2, Bool \)
\( p_2, Bool \)
\( p_3, Bool \)
\( p_3, Bool \)
\( p_3, Int \)
\( p_4, Int \)
\( p_4, Int \)

\( p_0, \tau \longrightarrow p_1, Int \)
\( p_1, \tau \longrightarrow p_2, \tau \)
\( p_1, \tau \longrightarrow p_3, \tau \)
\( p_2, \tau \longrightarrow p_1, Bool, \text{ if } \tau \sqsubseteq \text{Rational} \)
\( p_3, Int \longrightarrow p_4, Int \)
Constant propagation analysis

\[ p_0 : \ x = 1; \ y = 2; \]
\[ p_1 : \ \text{while} \ (x < y + z) \]
\[ \quad \ p_2 : \ x = x + 1; \]
\[ \quad \} \]
\[ p_3 : \ \text{exit} \]

where \( m + n \) is interpreted

\[
\begin{align*}
    k_1 + k_2 & \longrightarrow \text{sum}(k_1, k_2), \\
    \top & \neq k_i \neq \bot, i \in 1..2 \\
    \top + k & \longrightarrow \top \\
    k + \top & \longrightarrow \top
\end{align*}
\]

Abstract trace:

\[
\begin{array}{l}
    p_0, (\top, \top, \top) \\
    p_1, (1, 2, \top) \\
    p_2, (1, 2, \top) \\
    p_1, (2, 2, \top) \\
    p_2, (2, 2, \top) \\
    p_1, (3, 2, \top) \\
    p_3, (1, 2, \top) \\
    p_3, (2, 2, \top) \\
    ... \\
\end{array}
\]
An acceleration is needed for finite convergence

\[ p_0, \langle \top, \top, \top \rangle \]
\[ p_1, \langle 1, 2, \top \rangle \]
\[ p_2, \langle 1, 2, \top \rangle \]
\[ p_1, \langle 2, 2, \top \rangle \sqcup \langle 1, 2, \top \rangle = p_1, \langle \top, 2, \top \rangle \]
\[ p_2, \langle \top, 2, \top \rangle \]
\[ p_0 \]
\[ p_1 \]
\[ p_2 \]
\[ p_3 \]

Drawn as a data-flow analysis:

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

\[ p_0 : \ x = 1; \ y = 2; \]
\[ p_1 : \ \text{while} \ (x < 2 + z) \]
\[ \quad p_2 : \ x = x + 1; \]
\[ \} \]
\[ p_3 : \ \text{exit} \]
Array bounds (pre)checking

Integer variables receive values from the interval domain,

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

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

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

This generates an infinite sequence; we must accelerate with \(\triangledown\):

\[ P_1: \quad i = [0,0] \sqcup [-\infty,9] = [0,0] \]
\[ i = ([0,0] \triangledown [1,1]) \sqcup [-\infty,9] = [0, +\infty] \sqcup [-\infty,9] = [0,9] \]

\[ P_2: \quad i = [1,1] \triangledown [1,10] = [1, +\infty] \]
We wish to prove that $z \geq x \land z \geq y$ at $p_3$:

$p_0: \text{if } x < y$
$p_1: \text{then } z = y$
$p_2: \text{else } z = x$
$p_3: \text{exit}$

We chose three predicates, $\phi_1 = x < y$, $\phi_2 = z \geq x$, and $\phi_2 = z \geq y$ and computed their values at the program’s points. The predicates’ values come from the domain, $\{t, f, ?\}$. (Read $?$ as $t \lor f$.)

At all occurrences of $p_3$ in the abstract trace, $\phi_2 \land \phi_3$ holds.
When a goal is undecided, refinement is necessary

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

$$p_0 : \text{if } ! (x \geq y)$$

$$p_1 : \text{then } \{ i = x; $p_2 : x = y; $p_3 : y = i;$$ $p_4 : \}$$

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

These steps are justified by the following reasoning:

- Because $x \not\geq y$ and $x \geq i$ imply $y > i$ implies $x_{\text{new}} \geq i$
- Because $x \geq i$ implies $x_{\text{new}} \geq y$
But incremental predicate refinement cannot synthesize many interesting loop invariants. For this example:

\[
\begin{align*}
  p_0 & : \ i = n; \ x = 0; \\
  p_1 & : \ \text{while } i \neq 0 \{ \\
    p_2 & : \ x = x + 1; \ i = i - 1; \\
  \} \\
  p_3 & : \ \text{goal: } x = n
\end{align*}
\]

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 \( x = n - i \ : \)
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, $S_c \subseteq \mathcal{P}(\text{State})$, 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 $p', \phi_c$, that all such $p', s'$, where $s' \in \gamma(\phi_c)$, are “correct.”)
Every abstract domain defines a “logic”

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

$s$ has $a$, written $s \models_A a$, iff $\alpha\{s\} \subseteq a$

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

We have these facts: $p_1 \models_{PgmPt} p_1$ and $3 \models_{EqZero} \neg\text{zero}$
therefore, $p_1, 3 \models_{PP \times EqZero} p_1, \neg\text{zero}$
Abstract traces are examples of model checks

\[ p_0 : \text{while } x > 0 \{ \text{sleep} \} \]
\[ p_1 : \text{use resource} \]
\[ x = x + 1; \]  
\[ p_2 : \text{sleep} \]
\[ q_0 : x = 0; \]
\[ q_1 : \text{use resource} \]

Say that \( p_1, q_1 \) is an error state; is there a path from the start to it:
\[ p_0, q_0, \neg \text{zero} \]
\[ p_1, q_0, \neg \text{zero} \]
\[ p_1, q_1, \neg \text{zero} \]
\[ p_0, q_1, \neg \text{zero} \]
\[ p_1, q_1, \neg \text{zero} \]
\[ p_2, q_1, \text{zero} \]
\[ p_0, q_1, \text{zero} \]

Say that \( p_2 \) is a rest state; for all states reached from the start, can we progress to it:
\[ p_0, q_0, k \models LTL(PgmPt \times EqZero) F(p_1, q_1, \top) \]
\[ p_0, q_0, k \models LTL(PgmPt \times EqZero) GF(p_2, \top, \top) \]

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, \( \phi \), 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 \( \phi \).

**MiniLTL:** \( \phi ::= a \mid G\phi \mid F\phi \)

**Semantics:** \( [\phi] \subseteq \mathcal{P}(\text{Trace}) \)

\[
[a] = \{ \pi \mid \pi_0 \models \mathcal{A} a \}
\]

\[
[G\phi] = \{ \pi \mid \forall i \geq 0, \pi \downarrow i \in [\phi] \}
\]

\[
[F\phi] = \{ \pi \mid \exists i \geq 0, \pi \downarrow i \in [\phi] \}
\]

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, \( (\mathcal{P}(\text{Trace}), \subseteq) \leftrightarrow (\mathcal{P}(\text{MiniLTL}), \supseteq) \),

where \( \sqcup = \cap = \wedge \) in \( \mathcal{P}(\text{MiniLTL}) \):

\[
\gamma(P) = \cap \{ [\phi] \mid \phi \in P \}
\]

\[
\beta(\pi) = \{ \phi \mid \pi \in \gamma{\phi} \}
\]

\[
\alpha(S) = \cap \{ \beta(\pi) \mid \pi \in S \}
\]
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”)

\[= elephant_{species} \land brown_{color} \land heavy_{weight} \land 4000..6000kg_{weight} \land \cdots\]

There is even a pattern of Galois connection for this:

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

\[\beta : RealWorldObjects \rightarrow AllPossibleProperties\]
\[\beta(c) = \bigcap \{ p \in AllPossibleProperties \mid c \in \gamma(p) \}\]

\[\alpha : \mathcal{P}(RealWorldObjects) \rightarrow AllPossibleProperties\]
\[\alpha(S) = \bigcup \{ \beta(s) \mid s \in S \}\]
Some references

♦ The papers of Patrick and Radhia Cousot (www.di.ens.fr/~cousot), but especially


♦ A few of my papers, found at www.cis.ksu.edu/~schmidt/papers:
  2. Data-flow analysis is model checking of abstract interpretations. ACM POPL 1998.
  3. From Trace Sets to Modal-Transition Systems by Stepwise Abstract