Introduction to Abstraction and Static Analysis

David Schmidt
Kansas State University

www.cis.ksu.edu/~schmidt
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.**
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 $\gamma$ 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 $S \in \mathcal{P}($ConcreteData$)$, $a \in $AbstractProperties$, $S \subseteq \gamma(a)$ iff $\alpha(S) \subseteq a$. When $\alpha$ and $\gamma$ are monotone, this is equivalent to $S \subseteq \gamma \circ \alpha(S)$ and $\alpha \circ \gamma(a) \subseteq a$.

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];
...
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

```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

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

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

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

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

Two trace trees cover the full range of inputs:
The interpretation of the program’s semantics with the abstract values is an *abstract interpretation*:

\[
\begin{align*}
& \text{if } x \text{ is even-valued, } p_1, \text{ even } \\
& \text{if } x \text{ is odd-valued, the loop body, } p_1, \text{ will not be entered }
\end{align*}
\]

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 cannot decide termination for almost all the even-valued inputs. (Indeed, only \( 0 \) causes nontermination.)
The underlying abstract-interpretation semantics

\[ \gamma : \text{Polarity} \rightarrow \mathcal{P}(\text{Int}) \]
\[ \gamma(\text{even}) = \{\ldots, -2, 0, 2, \ldots\} \]
\[ \gamma(\text{odd}) = \{\ldots, -1, 1, 3, \ldots\} \]
\[ \gamma(\top) = \text{Int}, \quad \gamma(\bot) = \{\} \]

\[ \alpha : \mathcal{P}(\text{Int}) \rightarrow \text{Polarity} \]
\[ \alpha(S) = \sqcup \{\beta(v) | v \in S\}, \text{ where } \beta(2n) = \text{even} \text{ and } \beta(2n + 1) = \text{odd} \]

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,

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

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

Given a Galois connection, $\alpha, \gamma$, 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

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

\[
\begin{align*}
p_0, \tau & \rightarrow p_1, \text{Int} \\
p_1, \tau & \rightarrow p_2, \tau \\
p_1, \tau & \rightarrow p_3, \tau \\
p_2, \tau & \rightarrow p_1, \text{Bool}, \text{if } \tau \sqsubseteq \text{Rational} \\
p_3, \text{Int} & \rightarrow p_4, \text{Int}
\end{align*}
\]

Abstract trace:
\[
\begin{align*}
p_0, \text{Object} \\
p_1, \text{Int} \\
p_2, \text{Int} \\
p_1, \text{Bool} \\
p_3, \text{Int} \\
p_2, \text{Bool} \\
p_3, \text{Bool} \\
p_4, \text{Int}
\end{align*}
\]
**Constant propagation analysis**

$p_0 : \ x = 1; \ y = 2;$

$p_1 : \ \text{while} \ (x < y + z)$$
    \begin{align*}
    p_2 & : \ x = x + 1; \\
    \end{align*}$

$p_3 : \ exit$

where $m + n$ is interpreted

$k_1 + k_2 \rightarrow \text{sum}(k_1, k_2)$,

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

$\top + k \rightarrow \top$

$k + \top \rightarrow \top$

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

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

Abstract trace:

- $p_0, \langle \top, \top, \top \rangle$
- $p_1, \langle 1, 2, \top \rangle$

\[
\begin{align*}
    &\quad \\
    \downarrow & \\
    &\quad \\
    p_3, \langle 1, 2, \top \rangle
\end{align*}
\]

- $p_2, \langle 1, 2, \top \rangle$

\[
\begin{align*}
    &\quad \\
    \downarrow & \\
    &\quad \\
    p_3, \langle 2, 2, \top \rangle
\end{align*}
\]

- $p_2, \langle 2, 2, \top \rangle$

\[
\begin{align*}
    &\quad \\
    \downarrow & \\
    &\quad \\
    p_3, \langle 3, 2, \top \rangle
\end{align*}
\]

\[
\begin{align*}
    &\quad \\
    \downarrow & \\
    &\quad \\
    \ldots
\end{align*}
\]
An *acceleration* is needed for finite convergence

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

Drawn as a data-flow analysis:

\[ p_0, \langle T, T, T \rangle \]
\[ p_1, \langle 1, 2, T \rangle \]
\[ p_2, \langle 1, 2, T \rangle \]
\[ p_3, \langle T, 2, T \rangle \]

\[ p_1 \xrightarrow{1, 2, T} p_3 \]
\[ p_2 \xrightarrow{1, 2, T} p_3 \]
\[ p_2 \xrightarrow{2, 2, T} p_3 \]

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

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

Integer variables receive values from the *interval domain*,

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

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

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

At convergence, \(i\)'s ranges are

- at \(p_1\): \([0..9] \)
- at \(p_2\): \([1..10] \)
- at loop exit: \([1..10] \sqcap [10, +\infty] = [10, 10] \)

---

Escuela I / 27
Examples of relations between variables’ values

These Figures are from *Abstract Interpretation: Achievements and Perspectives* by Patrick Cousot, Proc. SSGRR 2000.
The papers of Patrick and Radhia Cousot (www.di.ens.fr/~cousot), including


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.