What is an Abstraction?

- A property from some domain

**Example Abstraction**

Concrete values: sets of integers

Abstract values

Concretization function $\gamma$ maps each abstract value to concrete values it represents

**Abstraction is Imprecise**

Concrete values: sets of integers

Abstract values

Abstraction function $\alpha$ maps each concrete set to the best abstract value
Composing $\alpha$ and $\gamma$

$\gamma \alpha$

Concrete values: sets of integers

Abstract values

Abstraction followed by concretization is sound but imprecise

$\alpha$ and $\gamma$ Form a Galois Insertion

- $\alpha$ and $\gamma$ are monotonic
  - Recall: $f$ is monotonic if $x \leq y \Rightarrow f(x) \leq f(y)$
  - Also called “order preserving”
- $S \subseteq \gamma(\alpha(S))$ 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
  - $e ::= i \mid e * e$

- Standard semantics of the program
  - $\text{Eval} : e \rightarrow \text{Int}$
  - $\text{Eval}(i) = i$
  - $\text{Eval}(e_1 * e_2) = \text{Eval}(e_1) \times \text{Eval}(e_2)$

Abstraction

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

<table>
<thead>
<tr>
<th>$\times$</th>
<th>$+$</th>
<th>$0$</th>
<th>$-$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$0$</td>
<td>$-$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$-$</td>
<td>$-$</td>
<td>$0$</td>
<td>$+$</td>
</tr>
</tbody>
</table>

$\text{AEval} : e \rightarrow \{-, 0, +\}$

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

$\text{AEval}(e_1 * e_2) = \text{AEval}(e_1) \times \text{AEval}(e_2)$
\textbf{Soundness}

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

\textbf{Another Approach to Soundness}

- Natural concretization function
\[
\begin{align*}
\gamma(+) &= \{ i \mid i > 0 \} \\
\gamma(0) &= \{ 0 \} \\
\gamma(-) &= \{ i \mid i < 0 \}
\end{align*}
\]
- Note: This presentation is slightly non-standard
  - Usually defined in terms of execution traces

\textbf{Soundness (cont’d)}

- Our abstraction is sound if
  - $\text{Eval}(e) \in \gamma(\text{AEval}(e))$
- Soundness proof: later

\textbf{Adding Unary Negation}

- $e ::= i \mid e \ast e \mid -e$
- $\text{Eval}(-e) = -\text{Eval}(e)$
- $\text{AEval}(e) = -\text{AEval}(e)$

\[
\begin{array}{c|cccc}
- & + & 0 & - \\
\hline
+ & 0 & - & + \\
0 & - & 0 & +
\end{array}
\]
- No problems
Adding Addition

- $e ::= i \mid e \cdot e \mid -e \mid e + e$

- $Eval(e_1 + e_2) = Eval(e_1) + Eval(e_2)$
- $AEval(e_1 + e_2) = AEval(e_1) + AEval(e_2)$

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>+</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>?</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Our abstract domain is not closed under addition

Solution

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

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>⊥</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

Other operations also need to handle $T$

Two Ways to Lose Information

- OK: Abstraction still precise enough
  - $Eval((5 * 5) + 6) = 31$
  - $AEval((5*5) + 6) = (+ \times +) \pm + = +$
    - 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
  - $Eval((1 + 2) + -3) = 0$
  - $AEval((1 + 2) + -3) = (+ \pm +) \pm - = +\pm - = T$
    - 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

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>⊥</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Adding Integer Division (cont’d)

- We need to extend other abstract operations to work on \( \perp \).
- Every operation involving \( \perp \) results in \( \perp \).
  - All operations are strict in \( \perp \):
    \[
    \begin{align*}
    \perp \times a &= \perp \\
    a \times \perp &= \perp \\
    \perp + a &= \perp \\
    a + \perp &= \perp \\
    a - \perp &= \perp
    \end{align*}
    \]

Abstraction and Concretization

- Concretization function \( \Upsilon \):
  \[
  \begin{align*}
  \Upsilon(\top) &= \text{all integers} \\
  \Upsilon(+) &= \{ i \mid i > 0 \} \\
  \Upsilon(0) &= \{ 0 \} \\
  \Upsilon(-) &= \{ i \mid i < 0 \} \\
  \Upsilon(\perp) &= \emptyset
  \end{align*}
  \]
- Abstraction function maps concrete values (sets of integers) to smallest valid abstract element
  \[
  \alpha(S) = \begin{cases} 
  \perp & \exists i \in S, i < 0 \\
  0 & \exists i \in S, i = 0 \\
  \perp & \exists i \in S, i > 0 \\
  \perp & \text{otherwise}
  \end{cases}
  \]

The Abstract Domain

- Look, Ma, a lattice!
- We’ve got:
  - A set of elements \( \{ \perp, +, 0, -, \top \} \)
  - A relation \( \leq \) that is
    - Reflexive
    - Anti-symmetric
    - Transitive
  - And
    - The least upper bound (lub, \( \sqcup \)) and greatest lower bound (glb, \( \sqcap \)) exists for any pair of elements
    - So it’s a lattice

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: \( \alpha \) and \( \Upsilon \) form a Galois insertion if
    - \( \alpha \) and \( \Upsilon \) are monotone
    - \( S \subseteq \gamma(\alpha(S)) \) or \( \text{id} \leq \gamma \alpha \) for any concrete set \( S \)
    - \( A = \alpha(\gamma(A)) \) or \( \text{id} = \alpha \gamma \) for any abstract element \( A \)
Soundness, Again

\[ \text{AEval} \{ \perp, +, 0, -, T \} \]
\[ \gamma \downarrow \alpha \]
\[ \text{Eval} \]
\[ i \in S \]

- Our abstraction is sound if
  - \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \)
- Soundness proof: next

Conditions for Correctness

- We can show that if
  - \( \alpha \) and \( \gamma \) form a Galois insertion
  - And abstract operations \( \text{op} \) are locally correct
    - \( \gamma(\text{op}(a_1, ..., a_n)) \supseteq \text{op}(\gamma(a_1), ..., \gamma(a_n)) \)
    - Note: We’ve extended \( \text{op} \) pointwise to sets
      - I.e., if \( S \) and \( T \) are sets, \( S + T = \{s + t | s \in S, t \in T\} \)
- Then the abstract interpretation is sound

Proof: Show \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \)

- By structural induction on expressions
  - Base cases: an integer \( i \), so \( \text{Eval}(i) = i \)
    - if \( i < 0 \) then \( \gamma(\text{AEval}(i)) = \gamma(\cdot) = \{ j | j < 0 \} \)
    - Other cases similar
  - Induction: for any operation
    \[ \text{Eval}(e_1 \text{ op } e_2) = \gamma(\text{AEval}(e_1)) \text{ op } \gamma(\text{AEval}(e_2)) \]
    - by definition of \( \text{AEval} \)
    by local correctness of \( \text{op} \)
    by definition of \( \text{AEval} \)

Another Proof of Correctness

- We can define correctness in terms of abstraction rather than concretization
  - \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \) iff \( \alpha(\{\text{Eval}(e)\}) \subseteq \text{AEval}(e) \)
- Equivalence proof:
  - (\( \Rightarrow \)) Assume \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \)
    - I.e., \( \{\text{Eval}(e)\} \subseteq \gamma(\text{AEval}(e)) \)
    - Then \( \alpha(\{\text{Eval}(e)\}) \leq \alpha(\gamma(\text{AEval}(e))) \) by monotonicity
    - And \( \alpha(\{\text{Eval}(e)\}) \leq \text{AEval}(e) \) since \( \text{id} = \alpha \gamma \)
Correctness Proof (cont’d)

- Showing
  - \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \) iff \( \alpha(\{\text{Eval}(e)\}) \leq \text{AEval}(e) \)
  - \( (\iff) \) Assume \( \alpha(\{\text{Eval}(e)\}) \leq \text{AEval}(e) \)
  - Then \( \gamma(\alpha(\{\text{Eval}(e)\})) \subseteq \gamma(\text{AEval}(e)) \) by monotonicity
  - Then \( \{\text{Eval}(e)\} \subseteq \gamma(\text{AEval}(e)) \) since \( \text{id} \leq \gamma \alpha \)
  - I.e., \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \)

An Alternate Abstract Domain

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

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

Acceleration: Widening

- Given monotone transfer functions
  - Finite height lattice \( \Rightarrow \) termination

- What if
  - Height is finite but large?
  - Height is infinite

- “Solution”: Widening
  - Every so often, replace \( A \) by \( A' > A \)
  - This is safe (conservative, sound)
  - But apply when? where?
Limitations

• Focus is on correctness
  - Not much insight into efficient algorithms

• Theory is completely general
  - What are good choices for modeling data structures and the heap? Higher-order functions? Objects?

• Forwards vs. backwards distinction
  - Permeates literature on abstract interpretation
  - But theory doesn’t require it

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