What is an Abstraction?

- A property from some domain

Example Abstraction

Concretes values: sets of integers

Abstract values

Concrete values: sets of integers

Abstract values

Composing $\alpha$ and $\gamma$

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
  - $e ::= i \mid e \ast e$

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

Abstraction

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

$$\begin{array}{c|ccc}
\times & + & 0 & - \\
+ & + & 0 & - \\
0 & 0 & 0 & 0 \\
\end{array}$$

- $\text{AEval} : e \rightarrow \{-, 0, +\}$
  - $\text{AEval}(i) = +$ if $i > 0$
  - $\text{AEval}(i) = 0$ if $i = 0$
  - $\text{AEval}(i) = -$ if $i < 0$
  - $\text{AEval}(e_1 \ast e_2) = \text{AEval}(e_1) \times \text{AEval}(e_2)$

Soundness

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

Soundness (cont’d)

- Natural concretization function
  - $\gamma(+) = \{i \mid i > 0\}$
  - $\gamma(0) = \{0\}$
  - $\gamma(-) = \{i \mid i < 0\}$

- Note: This presentation is slightly non-standard
  - Usually defined in terms of execution traces

Another Approach to Soundness

- Natural concretization function
  - $\gamma(+) = \{i \mid i > 0\}$
  - $\gamma(0) = \{0\}$
  - $\gamma(-) = \{i \mid i < 0\}$

- No problems

Adding Unary Negation

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

- $\gamma(+) = \{i \mid i > 0\}$
  - $\gamma(0) = \{0\}$
  - $\gamma(-) = \{i \mid i < 0\}$

- No problems
Adding Addition

- $e ::= i | e * e | -e | e + e$

- $\text{Eval}(e1 + e2) = \text{Eval}(e1) + \text{Eval}(e2)$

- $\text{AEval}(e1 + e2) = \text{AEval}(e1) + \text{AEval}(e2)$

Our abstract domain is not closed under addition.

Adding Integer Division

- $\text{div} + 0 - \text{div}$

<table>
<thead>
<tr>
<th>$\div$</th>
<th>$+$</th>
<th>$0$</th>
<th>$-$</th>
<th>$\text{div}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$-$</td>
<td>$-$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$\text{div}$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Our abstract domain is also not closed under division.

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>$\div$</th>
<th>$+$</th>
<th>$0$</th>
<th>$-$</th>
<th>$\text{div}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$0$</td>
<td>$0$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$-$</td>
<td>$-$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$\text{div}$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

- Other operations also need to handle $\text{div}$

The Abstract Domain

- Look, ma, a lattice!

- We've got:
  - A set of elements $\{\bot, +, 0, -, T\}$
  - A relation $\leq$ that is
    - Reflexive
    - Anti-symmetric
    - Transitive
  - And
    - The least upper bound ($\text{lub} \bot$) and greatest lower bound ($\text{glb} \bot$) exists for any pair of elements
    - So it's a lattice
Abstraction and Concretization

- Concretization function \( \gamma \)
  \[
  \begin{align*}
  \gamma(\top) &= \text{all integers} \\
  \gamma(+) &= \{ i | i \geq 0 \} \\
  \gamma(0) &= \{ 0 \} \\
  \gamma(-) &= \{ i | i < 0 \} \\
  \gamma(\bot) &= \emptyset
  \end{align*}
  \]
- Abstraction function maps concrete values (sets of integers) to smallest valid abstract elements
  \[
  \alpha(S) = \left\{ \begin{array}{ll}
  \bot & \text{if } S = \emptyset \\
  \{ 0 \} & \text{if } S = \{ 0 \} \\
  \{ i \} & \text{if } S = \{ i \} \\
  \{ 0, i \} & \text{if } S = \{ 0, i \} \\
  \{ i, j \} & \text{if } S = \{ i, j \} \\
  \{ i, j, k \} & \text{if } S = \{ i, j, k \} \\
  \{ i, j, k, l \} & \text{if } S = \{ i, j, k, l \} \\
  \{ i, j, k, l, m \} & \text{if } S = \{ i, j, k, l, m \} \\
  \{ i, j, k, l, m, n \} & \text{if } S = \{ i, j, k, l, m, n \} \\
  \end{array} \right.
  \]

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 \( \gamma \) form a Galois insertion if
  - \( \alpha \) and \( \gamma \) are monotone
  - \( S \subseteq \gamma(\alpha(S)) \) or \( \alpha \leq \gamma \) for any concrete set \( S \)
  - \( A = \alpha(\gamma(A)) \) or \( \alpha = \gamma \alpha \) for any abstract element \( A \)

Soundness, Again

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

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 \( \gamma(\text{AEval}(i)) = \gamma(-) = \{ j | j < 0 \} \)
    - Other cases similar
  - Induction: for any operation
    \[
    \begin{align*}
    \text{Eval}(e1 \ op \ e2) &= \text{Eval}(e1) \ op \ \text{Eval}(e2) & \text{by definition of Eval} \\
    &\in \gamma(\text{AEval}(e1)) \ op \ \gamma(\text{AEval}(e2)) & \text{by induction} \\
    &\subseteq \gamma(\text{AEval}(e1)) \ op \ \text{AEval}(e2)) & \text{by local correctness of \( op \)} \\
    &= \gamma(\text{AEval}(e1 \ op \ e2)) & \text{by definition of \( \text{AEval} \)}
    \end{align*}
    \]

Conditions for Correctness

- We can show that if
  - \( \alpha \) and \( \gamma \) form a Galois insertion
  - Abstract operations \( op \) are locally correct
    - \( \gamma(op(a1, ..., an)) \supseteq op(\gamma(a1), ..., \gamma(\gamma(an))) \)
    - Note: We’ve extended \( op \) 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

Another Proof of Correctness

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

• Showing
  - $\text{Eval}(e) \in \gamma(\text{AEval}(e))$ iff $\alpha([\text{Eval}(e)]) \leq \text{AEval}(e)$
  - $(\Rightarrow)$ 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 $\text{id} \leq \gamma \circ \alpha$
  - I.e., $\text{Eval}(e) \in \gamma(\text{AEval}(e))$

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