COMP 150-AVS

Abstract Interpretation

Based on lectures by David Schmidt and by Alex Aiken
What is an Abstraction?

- A property from some domain

- Blue (color)
- Planet (classification)
- 6000..7000km (radius)
Example Abstraction

Concrete values: sets of integers

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 \( \gamma \)

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

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

\[
\begin{align*}
\text{AEval}(i) &= \begin{cases} 
+ & i > 0 \\
0 & i = 0 \\
- & i < 0
\end{cases} \\
\text{AEval}(e_1 \times e_2) &= \text{AEval}(e_1) \times \text{AEval}(e_2)
\end{align*}
\]
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 \( \text{AEval}(e) = - \)
Another Approach to Soundness

- Natural concretization function

\[ \gamma(+) = \{i | i > 0\} \]
\[ \gamma(0) = \{0\} \]
\[ \gamma(-) = \{i | i < 0\} \]

- Note: This presentation is slightly non-standard
  - Usually defined in terms of execution traces
Soundness (cont’d)

- Our abstraction is sound if
  - \(\text{Eval}(e) \in \gamma(A\text{Eval}(e))\)

- Soundness proof: later
Adding Unary Negation

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

- $\text{Eval}(-e) = -\text{Eval}(e)$

- $\text{AEval}(e) = -\text{AEval}(e)$

- No problems
Adding Addition

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

- $\text{Eval}(e_1 + e_2) = \text{Eval}(e_1) + \text{Eval}(e_2)$

- $\text{AEval}(e_1 + e_2) = \text{AEval}(e_1) \pm \text{AEval}(e_2)$

\[
\begin{array}{c|cccc}
\pm & + & 0 & - \\
\hline
+ & + & + & ? \\
0 & + & 0 & - \\
- & ? & - & - \\
\end{array}
\]

Our abstract domain is not closed under addition
Solution

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

\[ \gamma(\top) = \{\text{integers}\} \]

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>\top</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>T</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>

• Other operations also need to handle \(\top\)
Two Ways to Lose Information

• OK: Abstraction still precise enough
  - $\text{Eval}((5 \times 5) + 6) = 31$
  - $\text{AEval}((5 \times 5) + 6) = (+ \times +) \pm + = +$
    - 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
  - $\text{Eval}((1 + 2) + -3) = 0$
  - $\text{AEval}((1 + 2) + -3) = (+ + +) \pm - = + + - = \top$
    - We also don’t know which value we computed
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

\[
\gamma(\bot) = \emptyset
\]

<table>
<thead>
<tr>
<th>(\div)</th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>\top</th>
<th>\bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>\top</td>
<td>\bot</td>
</tr>
<tr>
<td>0</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>\top</td>
<td>\bot</td>
</tr>
<tr>
<td>\top</td>
<td>\top</td>
<td>0</td>
<td>\top</td>
<td>\top</td>
<td>\bot</td>
</tr>
<tr>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
</tbody>
</table>
Adding Integer Division (cont’d)

• We need to extend other abstract operations to work on \( \bot \).

• Every operation involving \( \bot \) results in \( \bot \).
  
  ▪ All operations are *strict* in \( \bot \):

\[
\begin{align*}
\bot \times a &= \bot \\
a \times \bot &= \bot \\
\bot + a &= \bot \\
a + \bot &= \bot \\
-\bot &= \bot
\end{align*}
\]
The Abstract Domain

- Look, Ma, a lattice!
- We’ve got:
  - A set of elements \( \{\bot, +, 0, -, \top\} \)
  - 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
Abstraction and Concretization

• Concretization function $\gamma$

  $\gamma(\top) = \text{all integers}$
  $\gamma(+)= \{i \mid i>0\}$
  $\gamma(0)= \{0\}$
  $\gamma(-)= \{i \mid i<0\}$
  $\gamma(\bot)= \emptyset$

• Abstraction function maps concrete values (sets of integers) to smallest valid abstract element

  $\alpha(S) = \begin{cases} 
  - \exists i \in S . i<0 \\ 
  \bot \text{ otherwise} 
  \end{cases} \sqcup 
  \begin{cases} 
  0 \exists i \in S . i=0 \\ 
  \bot \text{ otherwise} 
  \end{cases} \sqcup 
  \begin{cases} 
  + \exists i \in S . i>0 \\ 
  \bot \text{ otherwise} 
  \end{cases}$
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 $\text{id} \leq \gamma \alpha$ for any concrete set $S$
  - $A = \alpha(\gamma(A))$ or $\text{id} = \alpha \gamma$ for any abstract element $A$
Soundness, Again

- 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, $S+T = \{s+t \mid s \in S, t \in T\}$

- 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 $\gamma(\text{AEval}(i)) = \gamma(-) = \{j | j < 0\}$
    - Other cases similar
  - Induction: for any operation
    
    $\text{Eval}(e_1 \text{ op } e_2)$
    $= \text{Eval}(e_1) \text{ op } \text{Eval}(e_2)$         by definition of $\text{Eval}$
    $\in \gamma(\text{AEval}(e_1)) \text{ op } \gamma(\text{AEval}(e_2))$ by induction
    $\subseteq \gamma(\text{AEval}(e_1) \text{ op } \text{AEval}(e_2))$ by local correctness of $\text{op}$
    $= \gamma(\text{AEval}(e_1 \text{ op } e_2))$         by definition of $\text{AEval}$
Another Proof of Correctness

- We can define correctness in terms of abstraction rather than concretization
  - \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \text{ iff } \alpha(\{\text{Eval}(e)\}) \leq \text{AEval}(e) \)

- Equivalence proof:
  - \((\Rightarrow)\) Assume \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \)
  - l.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 id = \( \alpha \gamma \)
Correctness Proof (cont’d)

• Showing
  - \( \text{Eval}(e) \in \gamma(\text{AEval}(e)) \) iff \( \alpha(\{\text{Eval}(e)\}) \leq \text{AEval}(e) \)
  
  - \((\leftrightarrow)\) 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

- The right domain depends on the problem we’re trying to solve
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 ⇒ 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)
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