Abstract Interpretation:
concrete and abstract semantics
Concrete semantics

- We consider a very tiny language that manages arithmetic operations on integers values.
- The (concrete) semantics of the languages can be defined by the function $\mu$ defined by:

$$e = i \mid e \ast e$$

$$\mu : Exp \rightarrow Int$$

$$\mu(i) = i$$

$$\mu(e_1 \ast e_2) = \mu(e_1) \times \mu(e_2)$$
Abstract Semantics

Consider now an abstract semantics over the domain of signs

\[ \sigma : \text{Exp} \rightarrow \{+, -, 0\} \]

\[ \sigma(i) = \begin{cases} + & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ - & \text{if } i < 0 \end{cases} \]

\[ \sigma(e_1 \cdot e_2) = \sigma(e_1) \times \sigma(e_2) \]
From a different perspective

- We can associate to each abstract value the set of concrete elements it represents.
- The concretization function:

\[ \gamma : \{+, 0, -\} \rightarrow 2^{\text{Int}} \]

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

- The concretization function $\gamma$ maps an abstract value to a set of concrete elements.
- Let $D$ denote the concrete domain and $A$ denote the abstract domain. The correctness of the abstract semantics wrt the concrete one can be expressed by:

$$\mu(e) \in \gamma(\sigma(e))$$
Abstract Interpretation

- Abstract Interpretation is:
  - Computing the semantics of a program in an abstract domain
  - In the case of signs, the domain so far is \{+,0,-\}.

- The abstract semantics should be correct
  - It is an over approximation of the concrete semantics

- The relation between the two domains is given by a concretization function
Consider the unary operator -

- Let us add to our language the unary operator -

\[ \mu(-e) = -\mu(e) \]

\[ \sigma(-e) = -\sigma(e) \]
Consider the binary operation $+\cdot$

- Adding the addition operator forces us to modify the domain, as the previous one is not able to represent the result of adding numbers of opposite sign

$$
\begin{align*}
\mu(e_1 + e_2) &= \mu(e_1) + \mu(e_2) \\
\sigma(e_1 + e_2) &= \sigma(e_1) + \sigma(e_2)
\end{align*}
$$

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

- We add to the domain a new element that represents all the integer numbers (both positive and negative, and zero)

\[ \gamma(T) = \text{Int} \]

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>O</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>O</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>
The operations should be revisited

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

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

Sometimes there is information loss due to the abstract operations

\[ \mu((1 + 2) + -3) = 0 \]
\[ \sigma((1 + 2) + -3) = (+ + +) + (+-+) = T \]

Sometimes there is no information loss, with respect to the abstraction

\[ \mu((5 * 5) + 6) = 31 \]
\[ \sigma((5 * 5) + 6) = (+ x +) + + = + \]
Consider the division operator /

- Problem: what is the result of dividing by zero? No number!
- So we need a new element in our domain that represents the empty set of integers (i.e. a failure state)

$$\gamma(⊥) = \emptyset$$

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

$\perp + x = \perp$

$\perp - = \perp$

$x \times \perp = \perp$

$- \perp = \perp$
The resulting abstract domain

- It is a finite complete lattice
- The partial order is coherent wrt the concretization function:

\[ x \leq y \iff \gamma(x) \subseteq \gamma(y) \]
The abstraction function

- The concretization function $\gamma$ has an adjoint function, the abstraction function $\alpha$.
- Function $\alpha$ maps a set of concrete values into the best representation of this set in the abstract domain (the smaller element of the abstract domain that represents of these elements).
- In our example:

$$\alpha : 2^{\text{Int}} \rightarrow A$$

$$\alpha(S) = \operatorname{lub}(\{\neg i < 0 \land i \in S\}, \{0 \mid 0 \in S\}, \{+ i > 0 \land i \in S\})$$

$$\sigma(i) = \alpha(\{i\})$$
An Abstract Interpretation consists of:
- An abstract domain $A$ and a concrete domain $D$
- $A$ and $D$ are complete lattices. Smaller means “more precise”
- Two monotone adjoint function that enjoy Galois insertion.
- Abstract operations that are correct wrt the concrete ones
- A fixpoint algorithm

Galois insertion:
\[
\forall x \in 2^D. \ x \subseteq \gamma(\alpha(x))
\]
\[
\forall a \in A. \ x = \alpha(\gamma(x))
\]
Correctness revisited

- If case of Galois insertion, these correctness conditions are equivalent (prove it!)

\[
\begin{align*}
    \mu(e) &\in \gamma(\sigma(e)) \\
    \sigma(e) &\geq \alpha(\{\mu(e)\})
\end{align*}
\]
Correctness

• We show that in order to ensure the correctness of the whole analysis the following conditions are sufficient:
  1. The function $\alpha$ and $\gamma$ are monotone
  2. The function $\alpha$ and $\gamma$ form a Galois insertion
  3. The abstract operations are locally correct, i.e.

\[
\gamma(\text{op}(s_1,\ldots,s_n)) \supseteq \text{op}(\gamma(s_1),\ldots,\gamma(s_n))
\]

• Notice that there is always a way to define a locally correct abstract operation. It is sufficient to consider the operations that returns the top element of the abstract domain.
Local correctness

\[ \gamma(\text{op}(s_1, \ldots, s_n)) \]

\[ \text{op(\(\gamma(s_1), \ldots, \gamma(s_n)\))} \]
Correctness proof

• We show by structural induction on e that:

\[ \mu(e) \in \gamma(\sigma(e)) \]

• Basic step:

\[
\begin{align*}
\mu(i) &= i \\
\in &= \{i\} \\
\subseteq &= \gamma(\alpha(\{i\})) \\
= &= \gamma(\sigma(i))
\end{align*}
\]
Correctness proof

Inductive Step

\[ \mu(e) \in \gamma(\sigma(e)) \]

\[
\begin{align*}
\mu(e_1 \text{ op } e_2) &= \mu(e_1) \text{ op } \mu(e_2) \\
\subseteq &\quad \gamma(\sigma(e_1)) \text{ op } \gamma(\sigma(e_2)) \\
= &\quad \gamma(\sigma(e_1 \text{ op } e_2))
\end{align*}
\]
Adding an input

• We can extend our tiny language with the possibility to get an input value from the user

• This means that we have a variable \( x \) in the expressions

\[
e = i \mid e \ast e \mid -e \mid \ldots \mid x
\]
Concrete semantics

• The semantic function $\mu$ becomes

$$\mu : \text{Exp} \rightarrow \text{Int} \rightarrow \text{Int}$$

• And we may express it in terms of a family of functions, having expressions as indeces and a single parameter (the input value)

$$\mu_i(j) = i$$
$$\mu_x(j) = j$$
$$\mu_{e_1 \cdot e_2}(j) = \mu_{e_1}(j) \cdot \mu_{e_2}(j)$$
$$\mu_{e_1 + e_2}(j) = \mu_{e_1}(j) + \mu_{e_2}(j)$$
$$\ldots = \ldots$$
Abstract semantics

• The same holds for the abstract semantic function $\sigma$

$$\sigma : \text{Exp} \rightarrow A \rightarrow A$$

• Also in this case we can express $\sigma$ by a family of functions:

$$\sigma_i(\overline{j}) = \overline{i}$$

$$\sigma_x(\overline{j}) = \overline{j}$$

$$\sigma_{e_1*e_2}(\overline{j}) = \sigma_{e_1}(\overline{j}) * \sigma_{e_2}(\overline{j})$$

$$\sigma_{e_1+e_2}(\overline{j}) = \sigma_{e_1}(\overline{j}) + \sigma_{e_2}(\overline{j})$$

$$\ldots = \ldots$$

$$\overline{i} = \alpha(\{i\})$$
Correctness

- The following conditions are equivalent

\[ \forall i. \mu_e(i) \in \gamma(\sigma_e(\alpha(\{i\}))) \]
\[ \mu_e \leq_D \gamma \circ \sigma_e \circ \alpha \]
\[ \alpha \circ \mu_e \leq_A \sigma_e \circ \alpha \]
Local correctness

- We can express the local correctness condition by:

\[
\sigma \circ \gamma \left( \sigma_{e_1}(j), \ldots, \sigma_{e_n}(j) \right) \subseteq \gamma \left( \sigma \circ \sigma_{e_1}(\tilde{j}), \ldots, \sigma \circ \sigma_{e_n}(\tilde{j}) \right)
\]
Conditional statement

\[ e = \ldots \ | \ if \ e = e \ then \ e \ else \ e \ | \ \ldots \]

- **Concrete semantics**
  \[ \mu_{if \ e_1 = e_2 \ then \ e_3 \ else \ e_4} (i) = \begin{cases} 
    \mu_{e_3} (i) & \text{if } \mu_{e_1} (i) = \mu_{e_2} (i) \\
    \mu_{e_4} (i) & \text{if } \mu_{e_1} (i) \neq \mu_{e_2} (i) 
  \end{cases} \]

- **Abstract semantics**
  \[ \sigma_{if \ e_1 = e_2 \ then \ e_3 \ else \ e_4} (\bar{i}) = \sigma_{e_3} (\bar{i}) \uplus \sigma_{e_4} (\bar{i}) \]

- Notice the role of the lub in the abstract domain
Correctness of the conditional statm.

• Assume that the condition is true (the other case is analogous)

\[
\begin{align*}
\mu_{e_3}(i) \\
\in \gamma(\sigma_{e_3}(\overline{i})) \\
\subseteq \gamma(\sigma_{e_3}(\overline{i})) \cup \gamma(\sigma_{e_4}(\overline{i})) \\
\subseteq \gamma\left(\sigma_{e_3}(\overline{i}) \cup \sigma_{e_4}(\overline{i})\right) \\
= \gamma(\sigma_{\text{if } e_1=e_2 \text{ then } e_3 \text{ else } e_4}(\overline{i}))
\end{align*}
\]
Recursion

- As a final step, we add recursive functions (on a single parameter)

\[
\text{program} = \text{def } f(x) = e \\
\text{e = ... } | f(e)
\]

- Until now, the concrete semantics was defined as:

\[
\mu : \text{Exp} \to \text{Int} \to \text{Int}_\bot
\]
Concrete semantics (function calls)

- In order to take into account the call of functions, the signature of $\mu$ becomes as follows:

\[
\mu' : \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}_\bot) \rightarrow \text{Int} \rightarrow \text{Int}_\bot
\]

\[
\mu'_f(e)(g)(j) = g(\mu'_e(g)(j))
\]

\[
\mu'_x(g)(j) = j
\]

\[
\mu'_{e_1+e_2}(g)(j) = \mu'_e_1(g)(j) + \mu'_e_2(g)(j)
\]
Semantics of recursive functions

\[ \mu' : \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}_\bot) \rightarrow \text{Int} \rightarrow \text{Int}_\bot \]

Consideriamo una funzione \( \text{def } f = e \)

Definiamo una catena ascendente \( f_0, f_1, \ldots \) in \( \text{Int} \rightarrow \text{Int}_\bot \)

\[
\begin{align*}
    f_0 &= \lambda x. \bot \\
    f_{i+1} &= \mu'_e(f_i)
\end{align*}
\]

Definiamo \( \mu_f = \bigcup_i f_i \)
def \( f = \) if \( x=0 \) then 1 else \( f(x - 1) \)

\[ f_0(i) = \bot \quad \text{for every } i \]

\[ f_1(i) = \mu' \text{ if } x=0 \text{ then 1 else } f(x - 1) \]

\[ (f_0)(i) = \]

\[ \begin{cases} 
\mu'_1(f_0)(i) = 1 & \text{if } i=0 \\
\mu'_{f(x - 1)}(f_0)(i) & \text{otherwise} 
\end{cases} \]

\[ = f_0(\mu'_{x - 1}(f_0)(i)) \]

\[ = f_0(\mu'_{x}(f_0)(i)) - f_0(\mu'_1(f_0)(i)) \]

\[ = f_0(i) - f_0(1) \]

\[ = \bot \]
def f = if x=0 then 1 else f(x - 1)

f_0(i) = ⊥ for each i

f_1(i) = 1 if i=0, ⊥ otherwise

f_2(i) = μ if x=0 then 1 else f(x - 1)(f_1)(i) =

\[
\begin{cases} 
\mu'(f_1)(i)=1 & \text{if } i=0 \\
\mu'_{f(x - 1)}(f_1)(i) & \text{otherwise}
\end{cases}
\]

= f_1 (μ'_{x - 1}(f_1)(i))

= f_1 (μ'_{x}(f_1)(i)) - f_1 (μ'_{1}(f_1)(i))

= ...
def f = if x=0 then 1 else f(x - 1)

\[ f_0(i) = \bot \text{ for every i} \]
\[ f_1(i) = 1 \quad \text{if } i=0, \quad \bot \text{ otherwise} \]
\[ f_2(i) = 1 \quad \text{if } i=0,1, \quad \bot \text{ otherwise} \]
\[ f_3(i) = 1 \quad \text{if } i=0,1,2, \quad \bot \text{ otherwise} \]
\[ f_4(i) = 1 \quad \text{if } i=0,1,2,3, \quad \bot \text{ otherwise} \]
...

\[ \mu(f) = \bigcup_{i \geq 0} f_i \]
Abstract Semantics

• In the same way, we need to extend the definition of the abstract semantics $\sigma$.
• We require that all the operations are monotone.

\[ \sigma' : \text{Exp} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A \]

\[ \sigma'_{f(e)}(g)(\bar{i}) = g(\sigma'_e(g)(\bar{i})) \]

\[ \sigma'_{x}(g)(\bar{i}) = \bar{i} \]

\[ \sigma'_{e_1+e_2}(g)(\bar{i}) = \sigma'_e(g)(\bar{i}) + \sigma'_e(g)(\bar{i}) \]
Abstract semantics of recursion

\[ \sigma': \text{Exp} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A \]

Consideriamo una funzione \( \text{def } f = e \)

Definiamo una catena ascendente \( \overline{f}_0, \overline{f}_1, \ldots \) in \( A \rightarrow A \)

\[ \overline{f}_0 = \lambda a. \perp \]
\[ \overline{f}_{i+1} = \sigma'_e(\overline{f}_i) \]

Definiamo \( \sigma_f = \bigcup_i \overline{f}_i \)
Correctness

- The abstract chain always covers the corresponding element in the concrete chain
Correctness (ctd.)

\[ \forall i. \ f_i(j) \in \gamma(\overline{f_i(j)}) \]
\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \in \bigcup_{i \geq 0} \gamma(\overline{f_i(j)}) \]
\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \in \gamma\left(\bigcup_{i \geq 0} \overline{f_i(j)}\right) \]
\[ \Rightarrow \mu_f(j) \in \gamma(\sigma_f(\overline{j})) \]
Summary

• Abstract Interpretation means
  – Define a concrete (collecting) semantics
  – Define an abstract semantics (domain and operations)
  – Prove the local correctness of the operations
  – Apply a fixpoint algorithm to compute the abstract semantics
  – Use a widening operator to ensure convergence (if the ascending chain condition does not hold)