Abstract Interpretation

Part 2

History

- One breakthrough paper
  - Cousot & Cousot 77 (?)
- Inspired by
  - Dataflow analysis
  - Denotational semantics
- Enthusiastically embraced by the community
  - At least the functional community . . .
  - At least the first half of the paper . . .

A Tiny Language

- Consider a language with only integers and multiplication.

\[ e = i \mid e * e \]

\[ \mu : \text{Exp} \to \text{Int} \]
\[ \mu(i) = i \]
\[ \mu(e_1 * e_2) = \mu(e_1) \times \mu(e_2) \]

An Abstraction

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

\[ \sigma(\text{Exp}) \to \{+, 0, -\} \]
\[ \sigma(i) = \begin{cases} + & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ - & \text{if } i < 0 \end{cases} \]
\[ \sigma(e_1 * e_2) = \sigma(e_1) \times \sigma(e_2) \]

Soundness

- We can show that this abstraction is correct in the sense that it correctly predicts the sign of an expression.
- Proof is by structural induction on \( e \).

\[ \mu(e) > 0 \iff \sigma(e) = + \]
\[ \mu(e) = 0 \iff \sigma(e) = 0 \]
\[ \mu(e) < 0 \iff \sigma(e) = - \]

Another View of Soundness

- The soundness proof is clunky
  - each case repeats the same idea.
- Instead, directly associate each abstract value with the set of concrete values it represents.

\[ \gamma : \{+, 0, -\} \to 2^{\mathbb{Z}} \]
\[ \gamma(+) = \{ i : i > 0 \} \]
\[ \gamma(0) = \{ 0 \} \]
\[ \gamma(-) = \{ i : i < 0 \} \]
Another View (Cont.)

- The concretization function
  - Mapping from abstract values to (sets of) concrete values
- Let
  - \( D \) be the concrete domain,
  - \( A \) the abstract domain.

\[
\begin{align*}
\mu(e) &\in \gamma(\sigma(e)) \\
\exp(\sigma) &\in 2^D
\end{align*}
\]

Abstract Interpretation

- This is an abstract interpretation.
  - Computation in an abstract domain
  - In this case \( (+, 0, -) \).
- The abstract semantics is sound
  - approximates the standard semantics.
- The concretization function establishes the connection between the two domains.

Adding -

- Extend our language with unary -

\[
\begin{align*}
\mu(-e) &= -\mu(e) &\begin{array}{c} \text{\texttt{-}} + \text{\texttt{0}} - \\
\text{\texttt{-}} + \text{\texttt{0}} + \end{array} \\
\sigma(-e) &= \overline{\sigma(e)}
\end{align*}
\]

Adding +

- Adding addition is not so easy.
- The abstract values are not closed under addition.

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

Solution

- We need another abstract value to represent a result that can be any integer.
- Finding a domain closed under all the abstract operations is often a key design problem.

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

Extending Other Operations

- We also need to extend the other abstract operations to work with \( T \).
Examples

Abstract computation loses information

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

No loss of information

\[ \mu((5\times5)+6) = 31 \]
\[ \sigma((5\times5)+6) = (+\times+)(\lnot+) = + + + + \]

Adding / (Integer Division)

- Adding / is straightforward except for the case of division by 0.
- If we divide each integer in a set by 0, what set of integers results? The empty set.

\[ \emptyset \]

\[ \{ 1, 2, 3 \} \]

Adding / (Cont.)

- As before we need to extend the other abstract operations.
- In this case, every entry involving bottom is bottom
  - all operations are strict in bottom

\[ \bot \uparrow \bot \]
\[ \bot \times \bot \]
\[ \bot \lnot \]

The Abstract Domain

- Our abstract domain forms a lattice.
  - A partial order \( x \leq y \iff y(x) \leq y(y) \)
  - Every finite subset has a least upper bound (lub) & greatest lower bound (glb).
- We write \( A \) for an abstract domain
  - a set of values + an ordering

\[ + \]

Lattice Lingo

- A lattice is complete if every subset (finite or infinite) has lub's and glb's.
  - Every finite lattice is complete
- Thus every lattice has a top/bottom element.
  - Usually needed in abstract interpretations.

The Abstraction Function

- The abstraction function maps concrete values to abstract values.
  - The dual of concreteization.
  - The smallest value of \( A \) that is the abstraction of a set of concrete values.

\[ a : 2^{\mathbb{N}} \rightarrow A \]
\[ a(S) = \text{lb}([-1<<0 \wedge i \in S],[0|0 \in S],[+1>>0 \wedge i \in S])] \]
**A General Definition**

- An abstract interpretation consists of:
  - An abstract domain $D$ and concrete domain $\mathcal{O}$
  - Concretization and abstraction functions forming a Galois insertion.
  - A (sound) abstract semantic function.

Galois insertion:

\[
\forall x \in \mathcal{O}. \ x \subseteq \gamma(\alpha(x)) \quad \text{id} \leq \gamma \circ \alpha \\
\forall a \in A. \ x = \alpha(\gamma(x)) \quad \text{or} \quad \text{id} = \alpha \circ \gamma
\]

**Galois Insertions**

- The abstract domain can be thought of as dividing the concrete domain into subsets (not disjoint).
- The abstraction function maps a subset of the domain to the smallest containing abstract value.

\[
id \leq \gamma \circ \alpha \\
id = \alpha \circ \gamma
\]

**Picture**

- In correct abstract interpretations, we expect the following diagram to commute.

\[
\begin{array}{c}
\sigma \\
\downarrow \gamma \leq \alpha \\
\mu \in \mathcal{O} \\
\end{array}
\]

**General Conditions for Correctness**

Three conditions guarantee correctness in general:

- $\alpha$ and $\gamma$ form a Galois insertion
  \[
id \leq \gamma \circ \alpha, \ \text{id} = \alpha \circ \gamma
\]
- $\alpha$ and $\gamma$ are monotonic
  \[
x \leq y \Rightarrow \alpha(x) \leq \alpha(y)
\]
- Abstract operations are locally correct:
  \[
  \gamma(\alpha(x_1, \ldots, x_n)) = \alpha(\gamma(x_1), \ldots, \gamma(x_n))
  \]

**Generic Correctness Proof**

Proof by induction on the structure of $e$:

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

\[
\begin{align*}
\mu(e_1 \ op \ e_2) &= \mu(e_1) \ op \ \mu(e_2) \quad \text{def. of } \mu \\
&\in \gamma(\sigma(e_1)) \ op \ \gamma(\sigma(e_2)) \quad \text{by induction} \\
&\subseteq \gamma(\sigma(e_1) \ op \ \sigma(e_2)) \quad \text{local correctness} \\
&= \gamma(\sigma(e_1 \ op \ e_2)) \quad \text{def. of } \sigma
\end{align*}
\]

**A Second Notion of Correctness**

- We can define correctness using abstraction instead of concretization.

\[
\begin{align*}
\mu(e) \in \gamma(\sigma(e)) &= \alpha(\mu(e)) \leq \sigma(e) \\
\Rightarrow \text{ direction} \\
\mu(e_1 \ op \ e_2) &\in \gamma(\sigma(e_1)) \\
\alpha(\mu(e)) &\leq \alpha(\gamma(\sigma(e_1))) \quad \text{monotonicity} \\
\alpha(\mu(e)) &\leq \sigma(e) \\
\alpha &\leq \gamma \circ \text{id}
\end{align*}
\]
Correctness (Cont.)

- The other direction...

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

\[ \Leftarrow \text{ direction} \]

\[ \sigma(\mu(e))) \leq \sigma(e) \]

- \( \gamma(\mu(e))) \leq \gamma(\sigma(e)) \) - monotonicity

\[ \mu(e) \in \gamma(\sigma(e)) \quad \text{id} \leq \gamma \]

---

Semantics

- The meaning function now has type

\[ \mu : \text{Exp} \rightarrow \text{Int} \rightarrow \text{Int} \]

- We write the function curried with the expression as a subscript.

\[ \mu_r(j) = i \]

\[ \mu_i(j) = j \]

\[ \mu_{\text{\#}}(j) = \mu_r(j) \circ \mu_i(j) \]

\[ \mu_{\text{\#}}(j) = \mu_r(j) \circ \mu_i(j) \]

\[ \ldots = \ldots \]

---

Abstract Semantics

- Abstract semantic function:

\[ \sigma : \text{Exp} \rightarrow A \rightarrow A \]

- Also write this semantics curried.

\[ \sigma_r(j) = i \]

\[ \sigma_i(j) = j \]

\[ \sigma_{\#}(j) = \sigma_r(j) \circ \sigma_i(j) \]

\[ \sigma_{\#}(j) = \sigma_r(j) \circ \sigma_i(j) \]

\[ \ldots = \ldots \]

\[ \sigma = \sigma(\langle i \rangle) \]

---

Correctness

- The correctness condition needs to be generalized.

- This is the first real use of the abstraction function.

- The following are all equivalent:

\[ \forall i, \mu_e(i) \in \gamma(\sigma_e(\alpha(i))) \]

\[ \mu_e \leq \gamma \circ \sigma_e \circ \alpha \]

\[ \alpha \circ \mu_e \leq \alpha \circ \sigma_e \circ \alpha \]

---

Local Correctness

- We also need a modified local correctness condition.

\[ \sigma(\gamma(\sigma_e(j)), \ldots, \gamma(\sigma_e(j))) \leq \gamma(\sigma(\sigma_e(j)), \ldots, \sigma_e(j))) \]
Proof of Correctness

\[ \text{Thm } \mu_e(j) = \gamma(\sigma_e(j)) \]

Proof (by induction)

- Basis: \( \mu_e(j) = j \in \gamma(\sigma_e(j)) \)
- Step: \( \mu_e(j) = j \in \gamma(\sigma_e(j)) \)

Step

- \( \mu_{e \rightarrow \alpha}(j) = \text{def of } \mu \)
- \( \gamma(\sigma_e(j) \cup \sigma_\alpha(j)) \text{ local correctness} \)
- \( \gamma(\sigma_e(j) \cup \sigma_\alpha(j)) \text{ def of } \sigma \)

If-Then-Else

\[ e = \text{... } | \text{ if } e = e \text{ then } e \text{ else } e \text{ ...} \]

\[ \mu_{e \rightarrow e \rightarrow \alpha} = \begin{cases} \mu_e(j) & \text{if } \mu_e(j) = \mu_\alpha(j) \\ \mu_\alpha(j) & \text{if } \mu_e(j) \neq \mu_\alpha(j) \end{cases} \]

\[ \sigma_{e \rightarrow e \rightarrow \alpha} = \sigma_e(j) \cup \sigma_\alpha(j) \]

- Note the lub operation in the abstract function: this is why we need lattices as domains.

Correctness of If-Then-Else

- Assume the true branch is taken.
- (The argument for the false branch is symmetric.)

\[ \mu_e(j) \]

- \( \in \gamma(\sigma_e(j)) \text{ by induction} \)
- \( \subseteq \gamma(\sigma_e(j) \cup \sigma_\alpha(j)) \)
- \( \subseteq \gamma(\sigma_e(j) \cup \sigma_\alpha(j)) \text{ monotonicity of } \gamma \)

Revised Meaning Function

- Define an auxiliary semantics taking a function (for the free variable \( f \)) and an integer (for \( x \)).

\[ \mu' : \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}_1) \rightarrow \text{Int} \rightarrow \text{Int}_1 \]

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

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

\[ \mu'_{f \rightarrow g}(j) = \mu'_f(g)(j) + \mu'_f(g)(j) \]

Meaning of Recursive Functions

- Define \( \mu : \text{Exp} \rightarrow \text{Int} \), \( \mu' : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int} \).

Consider a function \( \text{def } f = e \)

Define an ascending chain \( f_0, f_1, \ldots \) in \( \text{Int} \rightarrow \text{Int} \)

\( f_0 = \lambda x. 1 \)

\( f_{i+1} = \mu'_f(f_i) \)

Define \( \mu_i = \bigcup f_i \)
Abstract Semantics Revised

- Define an analogous auxiliary function for the abstract semantics.

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

\[ \sigma'_0(g)(\bar{i}) = g(\sigma'_0(g)(\bar{i})) \]

\[ \sigma'_1(g)(\bar{i}) = \gamma \]

\[ \sigma'_{n+1}(g)(\bar{i}) = \sigma'_0(g)(\bar{i}) + \sigma'_n(g)(\bar{i}) \]

Abstract Meaning of Recursion

\[ \sigma : \text{Exp} \rightarrow A \rightarrow A \]

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

Consider a function \( \text{def } f = e \)

Define an ascending chain \( \overline{F}_0, \overline{F}_1, \ldots \) in \( A \rightarrow A \)

\( \overline{F}_0 = \lambda a. \bot \)

\( \overline{F}_1 = \sigma(\overline{F}_1) \)

Define \( \sigma_i = \bigcup \overline{F}_i \)

Correctness

\[ f_1(j) \leq \gamma \]

\[ f_2(j) \leq \gamma \]

\[ f_3(j) \leq \gamma \]

Corresponding elements of the chain stand in the correct relationship.

Correctness (Cont.)

\[ \forall i. \; f_i(j) \in \gamma(\overline{F}_i(j)) \]

\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \subseteq \bigcup_{i \geq 0} \gamma(\overline{F}_i(j)) \quad \text{chains stabilize} \]

\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \subseteq \gamma \left( \bigcup_{i \geq 0} \overline{F}_i(j) \right) \quad \text{monotonicity of } \gamma \]

\[ \Rightarrow \mu_j(j) \in \gamma(\sigma_j(j)) \quad \text{by definition} \]

Example

\[ \text{def } f(x) = \begin{cases} 0 & \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x+1) \end{cases} \]

Abstraction:

\[ \text{Hfp}(\sigma' \mid \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x+1)) \]

Simplified:

\[ \text{Hfp}(A \overline{F}, A \overline{x} + \overline{x} \times \overline{F}(\overline{x} + \bot)) \]
Strictness

- We will assume our language is strict.
  - Makes little difference in quality of analysis for this example.
- Assume that \( f(\bot) = \bot \)
- Therefore it is sound to define \( \overline{f}(\bot) = \bot \)

Calculating the LFP

\[ T_1 = \begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array} \]

Notes

- In this case, the abstraction yields no useful information!
- Note that sequence of functions forms a strictly ascending chain until stabilization
  \( f_0 < f_1 < f_2 = f_3 = f_4 = \ldots \)
- But the sequence of values at particular points may not be strictly ascending:
  \( f_0(+) < f_1(+) = f_2(+) < f_3(+) = f_4(+) = \ldots \)

Notes (Cont.)

- Lesson: The fixed point is being computed in the domain \( (A \rightarrow A) \rightarrow A \rightarrow A \)
- The fixed point is not being computed in \( A \rightarrow A \)
- Make sure you check the domain of the fixed point operator.

Strictness Analysis

- In lazy functional languages, it may be desirable to change call-by-need (lazy evaluation) to call-by-value.
- CBN requires building “thunks” (closures) to capture the lexical environment of unevaluated expressions.
- CBV evaluates its argument immediately, which is wasteful (or even wrong) if the argument is never evaluated under CBN.
Correctness

- Substituting CBV for CBN is always correct if we somehow know that a function evaluates its argument(s).
- A function \( f \) is strict if \( f(\bot) = \bot \)
- Observation: if \( f \) is strict, then it is correct to pass arguments to \( f \) by value.

Outline

- Deciding whether a function is strict is undecidable.
- Mycroft’s idea: Use abstract interpretation.
- Correctness condition: If \( f \) is non-strict, we must report that it is non-strict.

The Abstract Domain

- Continue working with the same language (1 recursive function of 1 variable).
- New abstract domain 2:

\[
\begin{array}{c|c}
1 & \top \\
0 & \bot \\
\end{array}
\]

Concretization/Abstraction

- The concretization/abstraction functions say
  - 0 means the computation definitely diverges
  - 1 means nothing is known about the computation
  - \( D \) is the concrete domain

\[
\begin{align*}
\gamma(0) &= \{\bot\} \\
\alpha(\{\bot\}) &= 0 \\
\gamma(1) &= D \\
\alpha(D) &= 1 \text{ if } S \neq \{\bot\}
\end{align*}
\]

Abstract Semantics

- Next step is to define an abstract semantics
- Transform \( f : \text{Int} \rightarrow \text{Int} \) to \( \overline{\text{f}} : 2 \rightarrow 2 \)
- Transform values \( v : \text{Int} \) to \( \overline{v} : 2 \)
- To test strictness check if \( \overline{f}(0) = 0 \)

Abstract Semantics (Cont.)

- An \( a \) stands for an abstract value (0 or 1).
- Treat 0,1 as false, true respectively.

\[
\begin{align*}
\sigma_\gamma'(g)(a) &= a \\
\sigma_\alpha'(g)(a) &= 1 \\
\sigma_\gamma'(g)(a) &= \sigma_\gamma'(g)(a) \\
\sigma_\alpha'(g)(a) &= \sigma_\alpha'(g)(a) \land \sigma_\alpha'(g)(a) \\
\sigma_\gamma'(g)(a) &= g(\sigma_\gamma'(g)(a))
\end{align*}
\]
The Rest of the Rules

\[ \sigma_{\varphi_1}(g)(a) = \sigma_1(g)(a) \land \sigma_1(g)(a) \]
\[ \sigma_{\varphi_2}(g)(a) = \sigma_2(g)(a) \land \sigma_2(g)(a) \]
\[ \sigma_{\varphi_3} = \varphi_2 \sigma_2(g)(a) \land \sigma_2(g)(a) \land (\sigma_2(g)(a) \land \sigma_2(g)(a)) \]
\[ \sigma_{\text{def } f} = \varphi' \]

An Example

\[ \text{def } f(x) = \text{ if } x = 0 \text{ then } 1 \text{ else } x + f(x - 1) \]
\[ \text{Ifp}(\varphi'(f(x) = 0 \text{ then } 0 \text{ else } x + f(x - 1))) \]
\[ \lambda(\alpha, \alpha). \alpha \]
\[ (\lambda(\alpha, \alpha). \alpha). 0 = 0 \quad \text{The function is strict in } x \]

Calculating the LFP

\[ \Lambda f \Lambda x, x \rightarrow 1 \land \varphi(x, f(x)) \]
\[ \tau_0 = \begin{array}{c|c|c|c} x & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \end{array} \]
\[ \tau_1 = \begin{array}{c|c|c|c} x & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \end{array} \]
\[ \tau_2 = \begin{array}{c|c|c|c} x & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \end{array} \]

Another Example

\[ \text{def } f(x, y) = \text{ if } x = 0 \text{ then } 0 \text{ else } f(x - 1, f(x, y)) \]
\[ \text{Ifp}(\varphi'(f(x = 0 \text{ then } 0 \text{ else } f(x - 1, f(x, y))))) = \lambda(\alpha, \alpha). \alpha \]

Example (Cont.)

- For multi-argument functions, check each argument combination of the form \((1, 0, 1, 0, 0, 1)\).

\[ (\lambda(\alpha, \beta). \alpha)(0, 1) = 0 \quad x \text{ can be passed by value.} \]
\[ (\lambda(\alpha, \beta). \alpha)(1, 0) = 1 \quad \text{Unsafe to pass } y \text{ by value.} \]

Summary of Strictness Analysis

- Mycroft's technique is sound and practical.
  - Widely implemented for lazy functional languages.
  - Makes modest improvement in performance (a few %).
  - The theory of abstract interpretation is critical here.
- Mycroft's technique treats all values as atomic.
  - No refinement for components of lists, tuples, etc.
- Many research papers take up improvements for data types, higher-order functions, etc.
  - Most of these are very slow.
Conclusions

- The Cousot&Cousot paper(s) generated an enormous amount of other research.
- Abstract interpretation as a theory and abstract interpretation as a method of constructing tools are often confused.
- Slogan of most researchers:

Finite Lattices + Monotonic Functions = Program Analysis

Where is Abstract Interpretation Weak?

- Theory is completely general
- The part of the original paper people understand is limited
  - Finite domains + monotonic functions

Data Structures and the Heap

- Requires a finite abstraction
  - Which may be tuned to the program
  - More often is "empty list, list of length 1, unknown length"
- Similar comments apply to analyzing heap properties
  - E.g., a cell has 0 references, 1 references, many references

Size of Domains

- Large domains = slow analysis
- In practice, domains are forced to be small
  - Chain height is the critical measure
- The focus in abstract interpretation is on correctness
  - Not much insight into efficient algorithms

Context Sensitivity

- No particular insight into context sensitivity
- Any reasonable technique is an abstract interpretation

Higher-Order Functions

- Makes clear how to handle higher-order functions
  - Model as abstract, finite functions
  - Ordering on functions is pointwise
    - Problem: huge domains
- Break with the dependence on control-flow graphs
Forwards vs. Backwards

- The forwards vs. backwards mentality permeates much of the abstract interpretation literature.
- But nothing in the theory says it has to be that way.

Next Time: Type Inference

- Theory
  - More, esp. flavors of unification
- Examples
  - From ML type inference to alias analysis