Abstract Interpretation

Lecture 16

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.

\[
e = i \mid e \cdot e
\]

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

\[
\mu(i) = i
\]

\[
\mu(e_1 \cdot e_2) = \mu(e_1) \cdot \mu(e_2)
\]

An Abstraction

• Define an abstract semantics that computes only the sign of the result.

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

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

\[
\sigma(e_1 \cdot e_2) = \sigma(e_1) \cdot \sigma(e_2)
\]

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.

\[
y : (+, 0, -) \rightarrow 2^{\mathbb{Z}}
\]

\[
y(+) = \{ \frac{i}{i > 0} \}
\]

\[
y(0) = \{ i \}
\]

\[
y(-) = \{ \frac{i}{i < 0} \}
\]
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.

$$\mu(e) \in \gamma(o(e))$$

Abstract Interpretation

- This is an abstract interpretation.
  - Computation in an abstract domain
  - In this case $\{+,0,\cdot\}$.
- 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 $-$

$$
\begin{align*}
\mu(-e) &= -\mu(e) \\
o(-e) &= -o(e)
\end{align*}
$$

Adding $+$

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

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

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.

Extending Other Operations

- We also need to extend the other abstract operations to work with $T$.

$$
\begin{align*}
\gamma(T) = \text{Int}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c|ccc}
+ & 0 & T & T \\
\hline
0 & 0 & 0 & 0 \\
T & T & T & T \\
\end{array}
\end{align*}
$$
Examples

Abstract computation loses information

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

No loss of information

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

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.

The Abstract Domain

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

The Abstraction Function

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

\[ \alpha: \mathbb{Z}^* \to \mathbb{A} \]
\[ \alpha(S) = \text{lub}(\{-\lfloor i / 0 \land i \in S\}, \{0 \land 0 \in S\}, \{+\lfloor i / 0 \land i \in S\}) \]

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.
A General Definition

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

Galois insertion:

\[
\forall x \in 2^D, \ x \subseteq \gamma(\alpha(x)) \quad \text{id} \leq \gamma \circ \alpha \\
\forall a \in A, \ x = \alpha(\gamma(x)) \quad \text{or} \quad \text{id} = \alpha \circ \gamma
\]

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.

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

General Conditions for Correctness

Three conditions guarantee correctness in general:

\[\alpha \text{ and } \gamma \text{ form a Galois insertion} \]
\[
\text{id} \leq \gamma \circ \alpha, \ \text{id} = \alpha \circ \gamma
\]
\[\alpha \text{ and } \gamma \text{ are monotonic} \]
\[x \leq y \Rightarrow \alpha(x) \leq \alpha(y)\]

Abstract operations \( \alpha \) are locally correct:

\[
\gamma(\Exp(\mu_1,\ldots,\mu_n)) \supseteq \Exp(\alpha(\mu_1),\ldots,\alpha(\mu_n))
\]

Generic Correctness Proof

Proof by induction on the structure of \( e \): \( \mu(e) \in \gamma(\alpha(e)) \)

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

A Second Notion of Correctness

- We can define correctness using abstraction instead of concretization.

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

\[\Rightarrow \text{direction} \]
\[
\mu(e) \in \gamma(\alpha(e)) \\
\alpha(\mu(e)) \leq \alpha(\alpha(e)) \quad \text{monotonicity} \\
\alpha(\mu(e)) = \alpha(e) \quad \alpha \circ \gamma = \text{id}
\]
Correctness (Cont.)

- The other direction . . .

\[ \mu(e) \subseteq \gamma(\alpha(e)) \leq \alpha(\mu(e)) \]

\[ \Leftarrow \text{ direction} \]

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

\[ \gamma(\alpha(\mu(e))) \leq \gamma(\alpha(e)) \text{ monotonicity} \]

\[ \mu(e) \subseteq \gamma(\alpha(e)) \quad \text{id} \leq \gamma \circ \alpha \]

A Language with Input

- The next step is to add language features besides new operations.
- We begin with input, modeled as a single free variable \( x \) in expressions.

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

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.

\[ \mu_i(j) = i \]

\[ \mu_j(j) = j \]

\[ \mu_{a \ast b}(j) = \mu_a(j) \ast \mu_b(j) \]

\[ \mu_{a \ast b}(j) = \mu_a(j) \ast \mu_b(j) \]

\[ \ldots \]

Abstract Semantics

- Abstract semantic function:

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

- Also write this semantics curried.

\[ \alpha_i(j) = i \]

\[ \alpha_j(j) = j \]

\[ \alpha_{a \circ b}(j) = \alpha_a(j) \ast \alpha_b(j) \]

\[ \alpha_{a \circ b}(j) = \alpha_a(j) \ast \alpha_b(j) \]

\[ \ldots \]

\[ \tilde{i} = \alpha(0) \]

Correctness

- The correctness condition needs to be generalized.
- This is the first real use of the abstraction function.
- The following are all equivalent:

\[ \forall i. \mu_e(i) \subseteq \gamma(\alpha_e(\alpha(0))) \]

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

\[ \alpha \circ \mu_e \leq \alpha \circ \alpha_e \circ \alpha \]

Local Correctness

- We also need a modified local correctness condition.

\[ \alpha_p(\gamma(\alpha_e(j)) \ldots, \gamma(\alpha_e(j))) \leq \gamma(\alpha_p(\alpha_e(j)) \ldots, \alpha_e(j))) \]
**Proof of Correctness**

*Then \( \mu_i(j) \in \gamma(\alpha_i) \)*

Proof (by induction)

**Basis.** \( \mu_i(j) = i \in \gamma(\alpha_i) \)

**Step** \( \mu_{i+1}(j) = \gamma(\alpha_i) \)

- Def of \( \mu \)
- \( \gamma(\alpha_i) \) induction
- \( \gamma(\alpha_i) \) local correctness

**If-Then-Else**

\[
e = \ldots \mid \begin{array}{l}
e \mbox{ if } e \mbox{ then } e \mbox{ else } e \mid \ldots \\
\end{array}
\]

\[
\nu_{i+1}(e) = \begin{cases} 
\mu_i(e) & \text{if } \mu_i(e) = \mu_i(f) \\
\mu_i(f) & \text{if } \mu_i(e) \neq \mu_i(f) 
\end{cases}
\]

- 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.)

\[
\mu_i(e) \subseteq \gamma(\alpha_i) \quad \text{by induction}
\]

\[
\gamma(\alpha_i) \quad \gamma(\alpha_i) \quad \gamma(\alpha_i) \quad \text{monotonicity of } \gamma
\]

**Recursion**

- Add recursive definitions
  - of a single variable for simplicity
  - The semantic function is

\[
\mu : \text{Exp} \to \text{Int} \to \text{Int}_i
\]

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

\[
e = \ldots \mid f(e)
\]

**Revised Meaning Function**

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

\[
\mu : \text{Exp} \to (\text{Int} \to \text{Int}_i) \to \text{Int} \to \text{Int}_i
\]

\[
\mu_{i+1}(j) = \gamma(\alpha_i) \quad \mu_i(j) = j
\]

\[
\mu_{i+1}(g)(j) = \mu_i(g)(j) + \mu_i(g)(j)
\]

**Meaning of Recursive Functions**

\[
\mu : \text{Exp} \to \text{Int} \to \text{Int}_i
\]

\[
\mu : \text{Exp} \to (\text{Int} \to \text{Int}_i) \to \text{Int} \to \text{Int}_i
\]

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

Define an ascending chain \( \mu_{i+1} \) in \( \text{Int} \to \text{Int}_i \)

\[
\mu_{i+1} = \mu_i + e
\]

Define \( \mu_i = \bigcup f \)
**Abstract Semantics Revised**

- Define an analogous auxiliary function for the abstract semantics.

\[ \alpha': \text{Exp} \to (A \to A) \to A \to A \]

\[ \alpha'_f(g)\bar{\gamma} = g(\alpha'_v(g)\bar{\gamma}) \]

\[ \alpha'_v(g)\bar{\gamma} = \bar{\gamma} \]

\[ \alpha'_v(g)^{i+1} \bar{\gamma} = \alpha'_v(g)^i \bar{\gamma} + \alpha'_v(g)\bar{\gamma} \]

**Abstract Semantics Revised II**

- We need one more condition for the abstract semantics.

- All abstract functions are required to be monotonic.

- Thm. Any monotonic function on a complete lattice has a least fixed point.

**Abstract Meaning of Recursion**

- Define an ascending chain \( \bar{\gamma}_0, \bar{\gamma}_1, \ldots \) in \( A \to A \)

- Define \( \alpha_\gamma = \bigcup \bar{\gamma} \)

**Correctness**

- Corresponding elements of the chain stand in the correct relationship.

**Example**

- \( \text{def f(x) = if x = 0 then 1 else x + f(x - 1)} \)

- Abstract:
  \[ \text{lp}(\alpha'(\text{if x = 0 then 1 else x + f(x - 1)})) \]

- Simplified:
  \[ \text{lp}(\bar{x} \cup (x \times f(x \uparrow -1))) \]
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

\[
\begin{align*}
(\text{unsat}) &\Rightarrow (\text{unsat}) \\
(\text{unsat}) &\Rightarrow (\text{unsat}) \\
(\text{unsat}) &\Rightarrow (\text{unsat}) \\
(\text{unsat}) &\Rightarrow (\text{unsat}) \\
(\text{unsat}) &\Rightarrow (\text{unsat})
\end{align*}
\]

Notes

- In this case, the abstraction yields no useful information!
- Note that sequence of functions forms a strictly ascending chain until stabilization \( f_0 < f_1 < f_2 < f_3 = f_4 = f_5 = \ldots \)
- But the sequence of values at particular points may not be strictly ascending: \( f_0(+) < f_1(+) = f_2(+) < f_3(+) = f_4(+) = f_5(+) = \ldots \)

Notes (Cont.)

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

Strictness Analysis Overview

- 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:

\[
\begin{array}{c|c}
1 & 2 \\
\hline
0 & \end{array}
\]

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

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

Abstract Semantics

• Next step is to define an abstract semantics

• Transform \( f: \text{Int} \rightarrow \text{Int} \) to \( f : 2 \rightarrow 2 \)

• Transform values \( v: \text{Int} \) to \( v : 2 \)

• To test strictness check if \( f(0) = 0 \)

Abstract Semantics (Cont.)

• An \( a \) stands for an abstract value (0 or 1).

• Treat 0,1 as false, true respectively.

\[
\begin{align*}
\sigma'_f(g)(a) &= a \\
\sigma'_f(g)(a) &= 1 \\
\sigma'^f(g)(a) &= a' \sigma'_f(g)(a) \\
\sigma'^{\text{eq}}(g)(a) &= \sigma'_c(g)(a) \land \sigma'^{c}(g)(a) \\
\sigma'^c(g)(a) &= g(\sigma'_c(g)(a))
\end{align*}
\]
The Rest of the Rules

\[ o_{\varphi}(g)(x) = o_{\varphi}(g)(x) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]
\[ o_{\varphi}(g)(a) = o_{\varphi}(g)(a) \]

An Example

\[
\text{def } f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x & \text{else} \end{cases} + f(x - 1)
\]
\[ \text{lfp}(\lambda x. x) = 0 \]

Calculating the LFP

\[
\text{lfp}(\lambda f. \lambda x. x \cdot 1 \cdot (1 \lor (\lambda x. f(x + 1))))
\]
\[ \tau_0 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \]
\[ \tau_1 = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \]
\[ \tau_2 = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \]

Another Example

\[
\text{def } f(x,y) = \begin{cases} 0 & \text{if } x = 0 \\ f(x - 1, f(x,y)) & \text{else} \end{cases} + f(x + 1, f(x,y))
\]
\[ \text{lfp}(\lambda f. \lambda x. x \cdot 1 \cdot (1 \lor ...)) = \lambda(x,y). x \]

Example (Cont.)

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

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

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:

\[ \text{Finite Lattices + Monotonic Functions} = \text{Program Analysis} \]

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