Abstract Interpretation

Part II

Lecture 19

Review

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

Abstraction:

lfp(if x = 0 then 1 else x + f(x-1))

Simplified:

lfp(λf. x. x f(x))

Notes (Cont.)

• Lesson: The fixed point is being computed in the domain (A → A) → A → A

• The fixed point is not being computed in A → 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:

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

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

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

Abstract Semantics

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

Abstract Semantics (Cont.)

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

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

\[ \begin{align*}
\psi_{x^n}^l(\sigma_0) &= \psi_{x^n}^l(\sigma_0) \\
\psi_{x^n}^l(\sigma_0) &= \psi_{x^n}^l(\sigma_0) \\
\psi_{x^n}^l(\sigma_0) &= \psi_{x^n}^l(\sigma_0) \\
\psi_{x^n}^l(\sigma_0) &= \psi_{x^n}^l(\sigma_0)
\end{align*} \]

An Example

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

Calculating the LFP

\[
\text{lfp } \lambda f. x. x + 1 \text{ (if } x = 0 \text{ then 1 else } x + f(x - 1))
\]

Another Example

\[
\text{def } f(x, y) = \text{if } x = 0 \text{ then 0 else } f(x - 1, f(x, y))
\]

Example (Cont.)

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

Summary of Strictness Analysis

\[
\begin{align*}
\text{Mycroft's technique is sound and practical.} \\
\text{No refinement for components of lists, tuples, etc.} \\
\text{Many research papers take up improvements for data types, higher-order functions, etc.}
\end{align*}
\]
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

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
Context Sensitivity

Background

- Consider the program

\[ G(x) = \text{if } * \text{ then } M(0,x) \text{ else } M(x,0) \]

\[ M(a,b) = a * b \]

What is the rule-of-signs result for this program?

Strategy 1: Compute One Signature/Function

- Evaluate \( G(T,T) \)
  - Results in evaluating \( M(0,T) \)
  - Results in evaluating \( M(T,0) \)
  - Take upper bounds on all possible values of each argument
  - Final information \( M(T,T) = T \)
- A monomorphic analysis

Strategy 2: Inline \( M \)

\[ G(x) = \text{if } * \text{ then } 0 * x \text{ else } x * 0 \]

- Now we have \( G(T) = 0 \)
- Note that inlining
  - Separates the two uses of \( M \)
  - Allows different information to be assigned to each use

Strategy 3: Compute Signature for \( M \)

\[ G(x) = \text{if } * \text{ then } M(0,x) \text{ else } M(x,0) \]

\[ M(a,b) = a * b \]

- Analysis of \( M \) is

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

- Now use signature/table for \( M \) in analysis of \( G \)
- Same result as inlining

Discussion

- Monomorphic analysis is often imprecise
  - Combines all possible arguments arising at each use of function
  - Crucial issue is that correlations between arguments are lost
  - E.g., \( M(0,T) \) and \( M(T,0) \) become \( M(T,T) \)
- Many advantage is that it is cheap
  - Polynomial time
Discussion (Cont.)

- Inlining maintains correlations between function arguments
  - By analyzing every use of a function separately

- Using a precomputed signature for a function is similar to inlining
  - Usually more efficient, as function is analyzed once
  - For finite domains, just as precise
    - But could be less precise than inlining for more complicated abstract domains

Discussion (Cont.)

- Inlining is a poor choice in practice
  - Exponential blowup
  - Doesn’t even work in presence of recursion

- Two most common strategies
  - Reanalyze function for each distinct arguments
    - Moral equivalent of inlining
    - Exponential in time, not necessarily in space
  - Compute function summaries
    - Requires fixed points to deal with recursion

More Context Sensitivity

- Call site sensitivity is not the only kind of context sensitivity

- Consider the abstraction of allocation
  \textit{new C}

- Typically all values allocated at the same syntactic point share one abstract value

Example

\begin{verbatim}
Class X {
  int val;
  set(v) = val ← v; return this
  m(y) = return val × y
}

f(y) = return (new X).set(y)
\end{verbatim}

Example (Cont.)

Compare

\begin{verbatim}
(new X).set(0).m(z) + (new X).set(1).m(0)

f(0).m(z) + f(1).m(0)
\end{verbatim}

Summary

- Context-sensitivity is very important to accurate analysis of software
  - To maintain correlations between program quantities

- Unfortunately, solutions are generally exponential
  - In practice, systems either avoid it or pay a large performance price to have context sensitivity