Abstract Interpretation

Part II

Lecture 19

Review

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

Abstraction:

$$\text{lfp}\left(\sigma'\left(\text{if } x = 0 \text{ then } 1 \text{ else } x * f(x + \text{-}1)\right)\right)$$

Simplified:

$$\text{lfp}\left(\lambda\bar{f}.\lambda\bar{x}. + \cup\left(\bar{x} \;\bar{\times}\; \bar{f}(\bar{x} \;\bar{+}\; -)\right)\right)$$

Review: Calculating the LFP

$$\text{lfp}\left(\lambda\bar{f}.\lambda\bar{x}. + \cup\left(\bar{x} \;\bar{\times}\; \bar{f}(\bar{x} \;\bar{+}\; -)\right)\right)$$

$\bar{f}_0 =$

⊥	–	0	+	T
⊥	⊥	⊥	⊥	⊥

$\bar{f}_1 =$

⊥	–	0	+	T
⊥	+	+	+	+

$\bar{f}_2 =$

⊥	–	0	+	T
⊥	T	T	+	T

$\bar{f}_3 =$

⊥	–	0	+	T
⊥	T	T	T	T

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

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 $f(\perp) = \perp$
- 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:

$$\begin{array}{c} 1 \\ | \\ 0 \end{array}$$

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

$$\gamma(0) = \{\perp\} \qquad \alpha(\{\perp\}) = 0$$
$$\gamma(1) = D \qquad \alpha(S) = 1 \text{ if } S \neq \{\perp\}$$

Abstract Semantics

- Next step is to define an abstract semantics
- Transform $f\text{:Int} \to \text{Int}$ to $\bar{f}\text{:}2 \to 2$
- 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.

$$\begin{array}{lcl}
\sigma'_x(g)(a) & = & a \\
\sigma'_i(g)(a) & = & 1 \\
\sigma'_{-e}(g)(a) & = & \sigma'_e(g)(a) \\
\sigma'_{e_1 * e_2}(g)(a) & = & \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \\
\sigma'_{f(e)}(g)(a) & = & g(\sigma'_e(g)(a))
\end{array}$$

The Rest of the Rules

$$\begin{aligned}
\sigma'_{e_1+e_2}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a)\\
\sigma'_{e_1/e_2}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a)\\
\sigma'_{\text{if } e_1=e_2 \text{ then } e_3 \text{ else } e_4}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \wedge \left(\sigma'_{e_3}(g)(a) \vee \sigma'_{e_4}(g)(a)\right)\\
\sigma_{\text{def f} = e} &= \text{lfp } \sigma'_e
\end{aligned}$$

An Example

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

$$\text{lfp}\left(\sigma'\left(\text{if x} = 0 \text{ then } 1 \text{ else x} * \text{f(x} + -1)\right)\right)$$

$$\text{lfp}\left(\lambda\bar{f}.\lambda\bar{x}.\bar{x}\right) = \lambda a.a$$

$(\lambda a.a)\ 0 = 0$ The function is strict in x.

Calculating the LFP

$$\text{lfp}\left(\lambda\bar{f}.\lambda\bar{x}.\bar{x} \wedge 1 \wedge \left(1 \vee (\bar{x} \wedge \bar{f}(\bar{x} \wedge 1))\right)\right)$$

$$\bar{f}_0 = \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 0 \\ \hline\end{array}$$

$$\bar{f}_1 = \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 1 \\ \hline\end{array}$$

$$\bar{f}_2 = \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 1 \\ \hline\end{array}$$

Another Example

- Generalize to recursive functions of two variables.

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

$$\text{lfp}\left(\sigma'\left(\text{if x} = 0 \text{ then } 0 \text{ else f(x} + -1, \text{f(x,y))}\right)\right) =$$

$$\text{lfp}(\lambda\bar{f}.\lambda(\bar{x},\bar{y}).\ \bar{x} \wedge 1 \wedge (1 \vee \ldots)) =$$

$$\lambda(\bar{x},\bar{y}).\ \bar{x}$$

Example (Cont.)

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

$\left(\lambda(\bar{x},\bar{y}).\ \bar{x}\right)\ (0,1) = 0$ *X can be passed by value.*

$\left(\lambda(\bar{x},\bar{y}).\ \bar{x}\right)\ (1,0) = 1$ *Unsafe to pass Y 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

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) = if * then M(0,x) 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) = if * then 0 * x 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) = if * then M(0,x) else M(x,0)

M(a,b) = a * b

- Analysis of M is

×	+	0	–	T
+	+	0	–	T
0	0	0	0	0
–	–	0	+	T
T	T	0	T	T

- 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

  new C

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

Example

```
Class X {
    int val;
    set(v) =  val <- v; return this
    m(y) = return val * y
}

f(y) = return (new X).set(y)
```


Example (Cont.)

Compare

(new X).set(0).m(z) + (new X).set(1).m(0)

f(0).m(z) + f(1).m(0)

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