Announcements

- HW3: XTA Analysis
  - Due Thursday February 21st
  - Submitly for HW3 open
- HW1, Quiz1-2 are graded
  - If you got points off, stop by to get your marked paper
- Quiz 3

Outline

- Context sensitivity in practice, conclusion
- IFDS and CFL-reachability (optional slides)
- Abstract interpretation
- Semantics
- Notion of abstraction

Context-Sensitive Analysis In Practice

- Ad-hoc variants of Sharir and Pnueli’s call string and functional approaches
- Call string approach
  - More intuitive than functional approach
  - Virtually universally applicable, widely used
- Functional approach
  - Better approach, whenever applicable
  - Better precision and better scalability, in general
  - More difficult to implement
- Cloning-based approach

Call String-Based Context Sensitivity

- Calling context is defined as the content of the entire stack
- Call-string-based context-sensitivity uses a static call string as abstraction of the stack
- k-CFA: distinguishes context by k most recent call sites that lead to p
  - make a “copy” of procedure p for each static call string of length k
- 1-CFA: “inline” p at each call site of p

Example: 1-CFA
**PTA Example**

```java
A a = new A();  // o
X x = new X();  // o

c1: a.set(x);
A a2 = new B();  // o
X x2 = new Y(); // o

c2: a2.set(x2);
```

```java
main() {
    Context theContext = new Context();
    BoolExp or1 = new OrExp(new VarExp("X"),  // or1
                            new VarExp("Y"));
    BoolExp or2 = new OrExp(new Constant(true), // or2
                            new Constant(false));

    boolean result1 = or1.evaluate(theContext);
    boolean result2 = or2.evaluate(theContext);
}
```

**Boolean Expression Hierarchy:**

```java
public class OrExp extends BoolExp {
    private BoolExp left; private BoolExp right;

    public OrExp(BoolExp left, BoolExp right) {
        super(left, right);
    }

    public abstract class BinaryExp extends BoolExp {
        private BoolExp left; private BoolExp right;

        public BinaryExp(BoolExp left, BoolExp right) {
            this.left = left; this.right = right;
        }
    }

    public boolean evaluate(Context c) {
        private BoolExp l = this.left;
        private BoolExp r = this.right;
        return l.evaluate(c) || r.evaluate(c);
    }
}
```

**What If We Changed Boolean Expression Hierarchy? 1-CFA?**

```java
public abstract class BinaryExp extends BoolExp {
    private BoolExp left; private BoolExp right;

    public BinaryExp(BoolExp left, BoolExp right) {
        this.left = left; this.right = right;
    }
}
```
Cloning-based Context Sensitivity

- Remember, calling context is the content of the entire call stack.
- Cloning-based context sensitivity uses program state of interest as abstraction of the stack.
- Clone (i.e., copy) a procedure per “program state of interest”, i.e., “calling context”.
- A hybrid of functional and call-string.

Cloning-Based Context Sensitivity

- A a = new A(); // o1
  c1: a.set(new X()); // o2
  c2: a.set(new X()); // o3

- A a2 = new B(); // o4
  c3: a2.set(new Y()); // o5

// set(X p) { this.f = p; }

Cloning-Based Context Sensitivity

It is more effective if we “cloned” method set per receiver object rather than per call site.

A a = new A(); // o1
  c1: a.set_o1(new X()); // o2
  c2: a.set_o2(new X()); // o3

A a2 = new A(); // o4
  c3: a2.set_o4(new Y()); // o5

Again, flow-insensitive and context-sensitive, reaches our “ground truth”.

Summary-based Context Sensitivity

- Compute summary transfer functions
  - x = id(y) applies $\Phi_{id} = f_{exp}$, i.e., add x $\rightarrow$ a for each y $\rightarrow$ a (points-to for C example)
  - p(t) applies the “identity function” (Sharir and Pnueli’s Available expressions example)
  - a.set(x) “sets field f of all objects a points to to point to the objects x points to” (PTA example)

- Phase 1: compute summary transfer functions
  - Collapse into SCC on call graph, then compute summaries bottom up.

- Phase 2: propagate values into callees

Strongly-Connected Components

- p forms a SSC.
- Compute summary of p, treating SCC as single procedure.
- Summary of p says a*b is NOT available.
Summary-based Context Sensitivity

- For a class of lattices and transfer functions one can represent functions, and compute summary transfer functions efficiently!
  - See IFDS slides, a really nice analysis
  - Soot has an IFDS implementation

Outline

- Context sensitivity in practice, conclusion
- IFDS and CFL-reachability (optional slides)
- Abstract interpretation
- Semantics
- Notion of abstraction

Abstract Interpretation

- Patrick Cousot and Radhia Cousot, POPL'77

  - A general framework
  - Building static analyses
  - Reasoning about correctness of static analysis
  - Comparing static analyses
  - Combines ideas from dataflow analysis (monotone frameworks) and formal verification (axiomatic semantics)

Reading

- "Principles of Program Analysis" by Nielsen, Nielsen and Hankin, Chapter 3
  - Alex Salcianu's friendlier account of Chapter 3: https://web.eecs.umich.edu/~bchandra/courses/papers/Salcianu_AbstractInterpretation.pdf
  - Lecture notes by Xavier Rival, ENS

Overview

- Points-to analysis is an abstraction. Abstracts infinitely many heap objects \( h_i \) created at site \( i \) into a single \( o_i \).
- Program Execution: Points-to Analysis (PTA):
  - \( x \to h_j: A \)
  - \( x = y.n(z) \) passes value of \( z \) to parameter \( p \)
  - \( x = y.n(z) \) pts(\( z \)) \( \subseteq \) pts(\( p \))

Intuitively, Abstract Interpretation is about relating one Analysis to another Analysis
Small-step Operational Semantics

- Also called trace semantics, or concrete semantics, models program execution
- Memory state maps variables (V) to values (Z): \( \sigma : V \rightarrow Z \)
- Control state describes where we are
  - label \( \ell \) (note: we used the term program point)
- Describes transition \( \langle \eta \rangle, \sigma_{\ell 1} \rightarrow \langle \eta \rangle, \sigma_{\ell 2} \) (read: program executes statement at label \( \eta \) on current state \( \sigma_{\ell 1} \) transitioning to label \( \eta \) in state \( \sigma_{\ell 2} \))

A Simple Imperative Language: Operational Semantics

- Operational semantics of expressions:
  - \([n](\sigma) = n \) // constant \( n \) evaluates to \( n \)
  - \([x](\sigma) = \sigma(x) \) // variable \( x \) evaluates to the value \( n \) that \( x \) maps to in \( \sigma \)
- Assignment: \( \xi : x = E ; \xi : \ldots \)
  - \( \langle \eta \rangle, \sigma \rightarrow \langle \xi \rangle, \sigma[x \leftarrow [E_1]](\sigma) \)
- Assignment: \( \xi : x = E_1 ; E_2 ; \xi : \ldots \)
  - \( \langle \eta \rangle, \sigma \rightarrow \langle \xi \rangle, \sigma[x \leftarrow [E_1]](\sigma)\) \( \text{Op} [E_2](\sigma) \)

A Simple Imperative Language: Syntax (We’ve Seen This Before!)

- \( E ::= x \mid n \) // simple expression
- \( S ::= x = E \mid x = E \text{ Op } E \) // assignment
- \( | \text{ while } ( b ) \text{ Seq} \) // loop
- \( | \text{ if } ( b ) \text{ Seq} \text{ else } \text{ Seq} \) // conditional
- \( \text{Seq} ::= \{ S ; \ldots ; S ; \} \) // sequence

- \( V \) is the set of program variables, \( x \in V \)
- \( Z \) is the set of values variables take, \( n \in Z \)
**Collecting Semantics**

- Collects **all states** a program can have at a given label (i.e., program point)
  - E.g., x cannot ever be null at j
  - n is always greater than 100 at j

- Given a label, we are interested in a function $\sigma : \text{Labels} \rightarrow 2^\Sigma$
  - The set of all states a program can have at $l_i$

---

**Outline**

- Context sensitivity in practice, conclusion
- IFDS and CFL-reachability (optional slides)
  - Abstract interpretation
  - Semantics
  - Notion of abstraction

**Abstraction Example 1: signs**

- Concrete values: sets of integers
- Abstract values: signs

Lattice of signs:

- $\bot$ represents the empty set
- $+$ represents any set of positive integers
- $\emptyset$ represents set $\{0\}$
- $-$ represents any set of negative integers
- $T$ represents any set of integers
Abstraction Example 1: signs

Concrete space:
- A lattice!
- \{\ldots, -2, -1, 0, 1, \ldots\}
- \{-2, -1, 0\}
- \{0\}
- \{1, 2\}
- \{-2, -1\}
- \{0\}
- \{0, 1, 2, \ldots\}

Abstract space:
- A lattice!
- \{\ldots, 0, 1, 2, \ldots\}
- \{0\}
- \{0, 1\}
- \{0, 1, 2, \ldots\}

We use the abstraction relation to define abstract semantics, i.e., the execution of program statements over abstract elements.

If \(x\) is \(\ddagger\) and \(y\) is \(\ddagger\) then \(x + y\) is \(\ddagger\)

(Since \(\ddagger\) represents only sets of positive integers, then \(x\) and \(y\)'s values are positive integers, therefore \(x + y\) is a positive integer)

We can also use \(U\) and \(\cap\)

if \(x\) is \(\ddagger\) and \(y\) is \(\ddagger\) then \(x \cup y\) is \(\ddagger\)

How about if \(x\) is \(\ddagger\) and \(y\) is \(0\)?

then \(x \cup y\) is \(T\)

because only \(\{0, 1, 2, 3, \ldots\}\) \(\mapsto T\) holds

No other relation holds

In the abstract, we include negative integers in \(x \cup y\) (we lose precision!)

Refine the abstract space

- \(T\) represents any set of integers
- \(\perp\) represents any set of non-negative integers
- \(\ddagger\) represents any set of positive integers
- \(\ddagger\) represents any set of non-zero integers
- \(\emptyset\) represents any set of negative integers
- \(\emptyset\) represents any set of non-positive integers
- \(\emptyset\) represents any set of zero integers
- \(\emptyset\) represents the empty set

- \(\emptyset\) represents any set of positive integers
- \(\emptyset\) represents any set of negative integers
- \(\emptyset\) represents any set of non-negative integers
- \(\emptyset\) represents any set of non-positive integers
- \(\emptyset\) represents any set of non-zero integers

- \(\emptyset\) represents any set of non-zero integers
Abstraction Example 2: constants

- Concrete elements: elements of concrete lattice, \( c \in 2^Z \)
- Abstract elements: \( \bot, T, n \), where \( n \in Z \)
- Flat lattice:
- Abstraction relation:
  - \( c \vdash n \) if and only if \( c = \{ n \} \)
  - \( \bot \) represents the empty set
  - \( T \) represents any set of integers

Abstraction Example 3: intervals

- Concrete elements: elements of \( 2^Z \)
- Abstract elements: \( \bot, T \), intervals \([a,b]\) where \( a \in Z \cup \{-\infty\} \) and \( b \in Z \cup \{\infty\} \) and \( a \leq b \)
- Abstract semantics
  - If \( x \) is \([a_1,b_1]\) and \( y \) is \([a_2,b_2]\) then \( x + y \) is?
  - If \( x \) is \([a_1,b_1]\) and \( y \) is \([a_2,b_2]\) then \( x \cup y \) is?
  - If \( x \) is \([a_1,b_1]\) and \( y \) is \([a_2,b_2]\) then \( x \cap y \) is?