Announcements

- Soon Quiz 1-2, HW1 and HW2 will be in Rainbow Grades
  - Sorry for the delay

- HW3: XTA Analysis
  - Submit HW3 open
  - Questions?

Outline of Today’s Class

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

Reading

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981
- Dragon book, Chapter 12.1-3 Dragon book
- (IFDS) Thomas Reps, Susan Horwitz and Mooly Sagiv, “Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL’95

Classical Ideas and Results

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981
  - $S_{FA}(j) = S_{CS}(j) = \text{MORP}(j)$
  - A finite lattice of dataflow facts
  - Distributive transfer functions
  - No local variables, and no parameter passing

Two Approaches to Context-sensitive Analysis

- Functional approach
  - Computes summary transfer functions $\Phi_p$ that summarize effect of procedure $p$
  - in($\text{return } p$) = $\Phi_p$(in(call $p$)), in(entry $p$) = $V \text{ in(call } p$)
  - avoids propagation along exit $p$ --- $\text{return } p$ edge
- Call-string approach
  - Multiple call-string tagged solutions per program point $j$ in $p$: e.g., < { a+b }, c1 >, < { }, c1(c2) > at 6
  - Uses call-string tags at exit $p$ --- return $p$ edges to avoid propagation along unrealizable paths
**Functional Approach: Example**

1. \( a = \&x \)
2. call \( b = \text{id}(a) \)
3. return \( b = \text{id}(a) \)

**Call String Approach**

1. \( a = \&x \)
2. call \( b = \text{id}(a) \)
3. return \( b = \text{id}(a) \)

**Call String Approach**

- At exit nodes, propagate only matching call strings.
- \(<\{\text{ret} \rightarrow x\}, \{c_1\}>\), \(<\{\text{ret} \rightarrow y\}, \{c_2\}>\) at 9
- Propagate \(<\{\text{ret} \rightarrow y\}, \{c_2\}>\) to 6, thus, \(<\{d \rightarrow y\}, \{c_2\}>\) matches call string \(<\{d \rightarrow y\}, \{c_2\}>\)

**Outline of Today’s Class**

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

**Context-Sensitive Analysis In Practice**

- Transfer functions are not distributive
- Local variables, flow of values from actual arguments to formal parameters, and from return to left-hand-side (stack)
- Procedures have side effects (heap)
- Sometimes there is no call graph
  - Function pointers
  - Virtual calls
  - Functions as first-class values
**Context-Sensitive Analysis In Practice**

- 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
  - More difficult to implement
  - Better precision and better scalability, in general

**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**

1. `a = &x`
2. `p_c1 = a; call id_c1`
3. `return id_c1; b = ret_c1`
4. `z = *b + *b; c = &y;`
5. `p_c2 = c; call id_c2`
6. `return id_c2; d = ret_c2`
7. `entry id_c1`
8. `ret_c1 = p_c1`
9. `exit id_c1`
10. `entry id_c2`
11. `ret_c2 = p_c2`
12. `exit p_c2`

**Problems?**

```
main: id:
...
a = &x; int* id(int* p) {
c1: b = id(a); c3: return id_impl(p);
z = *b + *b; }
c = &y; ...
c2: d = id(c); int* id_impl(int* p) {
...
return p;
}
```

**Problems?**

- We can extend analysis to 2-CFA
  - There is a copy of id per call strings c1, and c2: id_[c1] and id_[c2]
  - There is a copy of id_impl per call strings c1,c3 and c2,c3: id_impl_[c1,c3] and id_impl_[c2,c3]
- We can extend to 3-CFA, 4-CFA, etc.

**Some Notes**

- Exponential growth
- Efficient data structures (Binary Decision Diagrams) can make full call string practical
- 2-CFA and 3-CFA are popular string lengths
- Recursion renders infinite call strings
- One approach with recursion
  - Collapse strongly connected components in call graph into one big blob. Analyze blob as single procedure, context-insensitively
  - Analyze acyclic call graph with full call string
Strongly-Connected Components

- A SCC.
- Treat calls within context-insensitively (thus concluding that \( a*b \) is not available at 4)
- Analyze \( p \) per \( c_1 \), other calling contexts if any

```
1. read a, b
2. call p
3. return p
4. t = a*b
print t
5. entry p
6. if a == 0
7. call p
8. return p
t = a*b
9. exit p
```

Recall: Points-to Analysis for Java (PTA)

- Saw in context of class analysis framework
- Context-insensitive, flow-insensitive analysis
- Syntax
  - Object allocation: \( a_i: x = \text{new } A /\!\!/ o_i \)
  - Assignment: \( x = y \)
  - Field Write: \( x.f = y \)
  - Field Read: \( x = y.f \)
  - Virtual call: \( c_i: x = y.m(z) \)

Example: 0-CFA vs. PTA

```java
public class A {
    public static void main() {
        X x1 = new X(); // o1
        A a1 = new B(); // o2
        x1.f = a1;
        A a2 = x1.f;
a2.m();
        x2 = new X(); // o3
        A a3 = new C(); // o4
        x2.f = a3;
        A a4 = x2.f;
a4.m();
    }
}
```

Recall: PTA

- Next, define the analysis semantics
  - Transfer functions (constraints) over syntax
    - E.g., Allocation \( x = \text{new } A /\!\!/ o_i \)
    - for each reachable method \( m \)
    - for each Allocation site \( x = \text{new } A /\!\!/ o_i \) in \( m \)
      - \( o_i \) \( \subseteq \) pts(\( x \))
    - Note: pts(\( x \)) denotes the points-to set of \( x \)
  - Natural progression: \( \text{RTA} \Rightarrow \text{XTA} \Rightarrow \text{0-CFA} \Rightarrow \text{PTA} \)
PTA Example

```java
A a = new A(); // o1
X x = new X(); // o2

c1: a.set(x);
A a2 = new B(); // o3
X x2 = new Y(); // o4

c2: a2.set(x2);

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

Boolean Expression Hierarchy: PTA

```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);
}
```

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;
    }

    public boolean evaluate(Context c) {
        return this.evaluate(c) || other.evaluate(c);
    }
}
```

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

```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);
}
```
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 for each program state of interest, i.e., “calling context”

```
A a = new A(); // o1
c1: a.set(new X()); // o2
c2: a.set(new X()); // o3
A a2 = new A(); // 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 = idfy) applies “add x—a for each y—a” (points-to for C example)
  - p() applies the “identity function” (Sharir and Pnueli’s Available expressions example)
  - a.set(x) “sets field f of all objects a points 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 SCC.
- Compute summary of p treating SCC as single procedure
- Summary of p says a*b is NOT available

```
main
  p
  1. read a, b
  2. call p
  3. return p
  4. t = a*b
  5. entry p
  6. if a == 0 then a = a - 1
  7. call p
  8. return p
  9. exit p
```
Summary-based Context Sensitivity

- For a class of lattices and transfer functions one can represent functions, and compute summary transfer functions efficiently!

- If you can do summary-based e.g. IFDS, go ahead

- Otherwise
  - C, ASM: 2-CFA or 3-CFA
  - OO (Java, C++): cloning-based per object