Interprocedural Analysis and Context Sensitivity, conclusion
Announcements

- Quiz 3
- HW3 and HW4?
- More office hours coming up starting tomorrow
So far on interprocedural analysis

- Interprocedural control-flow graph (ICFG)
  - Realizable paths
  - Meet over all realizable paths (MORP)
- Classical ideas in interprocedural analysis
  - Functional approach
  - Call string approach

- Reading
  - Chapter 12.1-3 Dragon book
int id(int p) {
    return p;
}

a = 5;
c1: b = id(a);
x = b*b;
c = 6;
c2: d = id(c);
Outline of Today’s Class

- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Summary-based context sensitivity

Reading
- Chapter 12.1-3 Dragon book
Transfer functions are not distributive

Local variables, flow of values from actual arguments to formal parameters, and from return to left-hand-side

Procedures have side effects!

Sometimes there is no call graph!

Function pointers, virtual calls, functions as first-class values

Parameter passing mechanisms
Context-Sensitive Analysis In Practice

- Ad-hoc adaptation of Sharir and Pnueli’s call string or functional approach
- Call-string-based approaches
  - More intuitive than functional one
  - Nearly universally applicable, widely used
- Functional approaches
  - More difficult to implement
  - Not always applicable
  - Better precision and better scalability, in general
Call-String-Based Context Sensitivity

- 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 call string of length k in the original program

- 1-CFA: “inline” p at each call site of p in the original program
Example: 1-CFA

1. $a = 5$
   
2. $p_{c1} = a$
call id_c1
   
3. return id_c1
   
   b = ret_c1
   
4. $z = b \times b$
c = 6
   
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 $p_{c2}$
main:

... 

a = 5;

c1: b = id(a);

z = b*b;

c = 6;

c2: d = id(c);

... 

id:

int id(int p) {

c3: return id_impl(p);
}

int id_impl(int p) {
    return p;
}

...
Problems with 1-CFA?

We can avoid imprecision with 2-CFA!
Problems with k-CFA?

- 1-CFA may not be enough
- Program size grows exponentially
- In practice, 2-CFA and 3-CFA are popular approaches
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: \( c_i : x = y.m(z) \)
Recall: PTA

- Next, define the analysis semantics
- Constraints over syntax
  - E.g., Allocation $x = \text{new} A \ // o_i$
    for each reachable method $m$
    for each Allocation site $i$: $x = \text{new} A \ // o_i \ in \ m$
    \[ \{ o_i \} \subseteq Pt(x) \]
  - Note: $Pt(x)$ denotes the points-to set of $x$
- Progression: RTA $\Rightarrow$ XTA $\Rightarrow$ 0-CFA $\Rightarrow$ PTA
Recall: PTA Constraints

\( a_i : x = \text{new } A \cup/ o_i \quad \{ o_i \} \subseteq \text{Pt}(x) \)

\( x = y \quad \text{Pt}(y) \subseteq \text{Pt}(x) \)

\( x.f = y \quad \text{for each } o \text{ in } \text{Pt}(x). \quad \text{Pt}(y) \subseteq \text{Pt}(o.f) \)

\( x = y.f \quad \text{for each } o \text{ in } \text{Pt}(y). \quad \text{Pt}(o.f) \subseteq \text{Pt}(x) \)

\( c_i : x = y.m(z) \)

\( \text{for each } o \text{ in } \text{Pt}(y) \)

\( \text{let } m'(\text{this},p,\text{ret}) = \text{resolve}(o,m) \text{ in } \)

\( \rightarrow \{ o \} \subseteq \text{Pt}(\text{this}) \)

\( \rightarrow \text{Pt}(z) \subseteq \text{Pt}(p) \quad \text{Pt}(\text{ret}) \subseteq \text{Pt}(x) \)
public class A {
    public static void main() {
        X x1 = new X();    // o₁
        A a1 = new B();   // o₂
        x1.f = a1;  // o₁.f points to o₂
        A a2 = x1.f;  // a2 points to o₂
        a2.m();
    }
}

X x2 = new X();    // o₃
A a3 = new A();    // o₄
→ x2.f = a3;
    a4 = x2.f;
0-CFA vs. PTA Example

public class A {
    public static void main() {
        X x1 = new X();   // o_1
        A a1 = new B();   // o_2
        x1.f = a1;        // o_1.f points to o_2
        A a2 = x1.f;      // a2 points to o_2
        a2.m();
        // PTA: a2: { B }  
        // 0-CFA: a2: { B, C }

        X x2 = new X();   // o_3
        A a3 = new C();   // o_4
        x2.f = a3;        // o_3.f points to o_4
        A a4 = x2.f;      // a4 points to o_4
        a4.m();
    }
}

CSCI 4450/6450, A Milanova
Another PTA Example

X x1 = new X();  // o₁
A a1 = new B();  // o₂
c1: x1.set(a1);
X x2 = new X();  // o₃
A a2 = new C(); // o₄
c2: x2.set(a2);

// set(X p) { this.f = p; }
1-CFA PTA Example

X x1 = new X();  // o₁
A a1 = new B();  // o₂

c1: x1.set(a1);

X x2 = new X();  // o₃
A a2 = new C();  // o₄

c2: x2.set(a2);

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

this.c1.f = p.c₁ and this.c2.f = p.c₂
main() {
    Context theContext = new Context();

    BoolExp or1 = new OrExp(new VarExp("X"),
                            new VarExp("Y"));       // or1

    BoolExp or2 = new OrExp(new Constant(true),
                            new Constant(false));    // or2

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

Boolean Expression Hierarchy:
public class OrExp extends BoolExp {
    private BoolExp left; private BoolExp right;

    public OrExp(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);
    }
}

Boolean Expression Hierarchy:
PTA
Pt(this) = \{ \text{or}_1, \text{or}_2 \}
Pt(left) = \{ v_1, c_1 \}, Pt(right) = \{ v_2, c_2 \}
Pt(\text{or}_1.left) = Pt(\text{or}_2.left) = \{ v_1, c_1 \} !!!
Pt(l) = \{ v_1, c_1 \}
Pt(r) = \{ v_2, c_2 \}
Boolean Expression Hierarchy:
1-CFA
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 class OrExp extends BinaryExp {
    public OrExp(BoolExp left, BoolExp right) {
        c5: BinaryExp.BinaryExp(left,right); // call to super
    }
}
```
main() {
    Context theContext = new Context();

    c1: BoolExp or1 = new OrExp(new VarExp("X"),
                    new VarExp("Y")); // or1

    c2: BoolExp or2 = new OrExp(new Constant(true),
                    new Constant(false)); // or2

    c3: boolean result1 = or1.evaluate(theContext);
    c4: boolean result2 = or2.evaluate(theContext);
}

What If We Changed Boolean Expression Hierarchy: 1-CFA?
What If We Changed Boolean Expression Hierarchy: 1-CFA?
2-CFA?
Outline of Today’s Class

- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Summary-based context sensitivity

- Reading
  - Chapter 12.1-3 Dragon book
Summary-based Context
Sensitivity

- Compute summary transfer functions
  - $x = \text{id}(y)$ applies “add $x \rightarrow a$ for each $y \rightarrow a$” (points-to for C example)
  - $p()$ applies the “identity function” (Sharir and Pnueli’s Available expressions example)
  - $\text{a.set}(x)$ “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 SCC.
- Compute summary of $p$, treating SCC as single procedure
- Summary of $p$ says $a*b$ is NOT available 😞

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

- Context-sensitive analysis is difficult
- Different approaches
  - Call-string-based, also known as k-CFA
    - 2-CFA and 3-CFA
    - Intuitive, easier to implement
  - Summary-based
    - Harder to design and harder to implement
    - Generally, more precise and more scalable