Interprocedural Analysis and Context Sensitivity
Announcements

- Quiz 3 on Friday
- HW2 graded
So Far

- Flow-insensitive, context-insensitive analyses for Java
  - RTA
  - XTA
  - 0-CFA
  - PTA

- Context-sensitive analysis
Outline of Today’s Class

- Interprocedural control-flow graph (ICFG)
- Classical ideas in interprocedural analysis
- Context-sensitive analysis in practice
  - Notion of calling context
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity

- Reading
  - Chapter 12.1-3 Dragon book
Interprocedural Control Flow Graph (ICFG)

- Add procedure **entry** node and **exit** node
- At each procedure call add
  - A **call** node and a **call-entry** edge

```
2.call --> 7.entry
```

- A **return** node and an **exit-return** edge

```
3.return <-- 9.exit
```
int* id(int* p) {
    return p;
}

a = &x;

c1: b = id(a);
z = *b + *b;
c = &y;
c2: d = id(c);

1. a = &x
2. call id
3. return id
4. z = *b + *b
   c = &y
5. call id
6. return id
7. entry id
8. ret = p
9. exit id
Context-Insensitive Analysis

- Add explicit assignments at call and return
  - E.g., \( x = \text{id}(y) \)
  - \( p = y \) models flow from actual argument \( y \) to formal parameter \( p \)
  - \( x = \text{ret} \) models flow from return to left-hand-side

- Treat ICFG as one big CFG
  - Can be flow-sensitive or
  - Flow-insensitive
    - E.g., Andersen’s points-to analysis for C
Andersen’s Analysis for C

```c
int* id(int* p) {
    return p;
}
```

c1: b = id(a);
```
z = *b + *b;
```
c2: d = id(c);
```
a = &x;
```
```
1. a = &x
```
```
2. p = a
    call id
```
```
3. return id
    b = ret
```

4. z = *b + *b
    c = &y
```
```
5. p = c
    call id
```

6. return id
    d = ret
```
```
7. entry id
```
```
8. ret = p
```
```
9. exit id
```
int* id(int* p) {
    return p;
}

a = &x;

1. a = &x
2. p = a
3. return id
4. z = *b + *b
5. p = c
6. return id
7. entry id
8. ret = p
9. exit id

c1: b = id(a);
    z = *b + *b;
    c = &y;

c2: d = id(c);
Context-Insensitive Analysis

- Problem: merges data from different contexts

- Goal of context-sensitive analysis: track “realizable paths”
int* id(int* p) {
    return p;
}

a = &x;
c1: b = id(a);
z = *b + *b;
c = &y;
c2: d = id(c);
Another Example

```c
int fib(int z, int u) {
    if (z<3) {
        return u+1; /* ret = u+1; */
    } else {
        auxiliary variable ret
        holds the return values */
        c2: v = fib(z-1,u);
        c3: return fib(z-2,v)
    }
}
...

c1: y = fib(x,0);
...
```

What does `fib` compute? Here `z` and `u` are formal parameters; `ret` is the special variable holding the return value.
Another Example

main:
1)

fib:
4.entry
5.z<3
6. ret=u+1
7.exit
8.call
9.return
10.call
11.return

z=x
u=0
y=ret
z=z-1
u=u
v=ret
z=z-2
u=v
ret=ret
c1
c2
c3

Spring 21 CSCI 4450/6450, A Milanova
Realizable Paths (RP)

- Context-free grammar!
- Same-level (balanced) path (SLP):
  \[ M ::= e \quad e \text{ denotes intra-procedural edge} \]
  \[ \mid (c_i M)_{c_i} \text{ captures path from call to return} \]
  \[ \mid M M \]
  - An intra-procedural edge is annotated with \( e \)
  - Call-entry edge that originates at call site \( c_i \) is \( (c_i \)
  - Corresponding exit-return edge is \( )_{c_i} \)
- A path \( p \), from \( m \) to \( n \), is in \( SLP_{m,n} \) iff string along \( p \) is in language described by \( M \)
Realizable Paths (RP)

- Paths from 1 to any node \( n \) in program
- Another grammar, describes paths with outstanding calls (i.e., calls not yet returned):
  \[
  C ::= (c_1 | M (c_1 | C M)
  \]
- A path from entry node 1 to node \( n \) is in \( \text{RP}_{1,n} \) iff the string from 1 to \( n \) is in the language generated by either \( M \) or \( C \)
  - E.g., in Points-to example, 1,2,7,8,9,3 is in \( \text{RP}_{1,3} \) but 1,2,7,8,9,3,4,5,7,8,9,3 is NOT in \( \text{RP}_{1,3} \)
Is $p_1 = 1, 2, 4, 5, 6, 7$ in $RP_{1,7}$?

Is $p_2 = 1, 2, 4, 5, 8, 4, 5, 6, 7, 3$ in $RP_{1,3}$?
Meet Over All Realizable Paths (MORP)

- MORP \( (n) = \bigvee f_{n_k} \circ f_{n_{k-1}} \circ \cdots \circ f_{n_2} \circ f_1 \text{(init)} \)
  
  \( p=(1,n_2 \ldots n_k,n) \) is a path in \( \text{RP}_{1,n} \)

(\( \circ \) denotes function composition)

- Also called MVP (meet over all valid paths) or just MRP

- \( \text{MORP}(n) \leq \text{MOP}(n) \). Why?

- May be undecidable even for lattices of finite height

- Goal: encode context and restrict analysis over realizable paths, as much as possible
Outline of Today’s Class

- Interprocedural control-flow graph (ICFG)
- Classical ideas in interprocedural analysis
- Context-sensitive analysis in practice
  - Notion of calling context
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity

- Reading
  - Chapter 12.1-3 Dragon book
Classic Ideas and Results

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981
  - Amir Pnueli, Turing Award in 1996 for “For seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.”

- A finite lattice of dataflow facts
- Distributive transfer functions
- No local variables, no parameter passing
1. read a, b
   t = a*b

2. call p

3. return p

4. 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
Sharir and Pnueli Example

- Expression $a*b$ is NOT available at 4 if we consider _all_ paths
  - E.g., along 1,2,5,6,7,5,6,9,3,4 $a*b$ gets “killed” due to $a = a - 1$, and it is not recomputed

- Expression $a*b$ is available at 4 if we consider only realizable paths
  - Path 1,2,5,6,7,5,6,9,3,4 is unrealizable because return edge 9,3 does not match the call edge 7,5
  - 1,2,5,6,7,5 ... 9,8,9 ...
  - We know “kill” 6,7 is succeeded by 7,5, which must be balanced with 9,8, which is succeeded by “gen”
Functional Approach

- Operates on unchanged property space
- Computes summary transfer functions $\Phi_p$ that summarize the effect of procedure $p$

- Reduces problem to intraprocedural case:
  - $\text{in}(\text{return } p) = \Phi_p(\text{in}(\text{call } p))$
  - thus, avoids propagation from callee along the exit $p$ --- $\text{return } p$ edge!
Functional Approach

Phase 1:
Compute a **summary transfer function** $\Phi_p$ that captures effect of $p$. In example $\Phi_p$ is the **identity function**: nothing gets generated and nothing gets killed (simplifying a bit)

1. read $a$, $b$
   
   $$t = a \times b$$

2. call $p$

3. return $p$

4. $t = a \times b$
   
   print $t$

5. entry $p$

6. if $a == 0$ then
   
   $a = a - 1$

7. call $p$

8. return $p$

   $$t = a \times b$$

9. exit $p$
1. read a, b
   \[ t = a * b \]
2. call p
3. return p
4. \[ t = a * b \]
   print t
5. entry p
6. if a == 0 then
   a = a - 1
   call p
   return p
   \[ t = a * b \]
7. call p
8. return p
9. exit p

Phase 2:
Dataflow analysis:
• At return p
  \[ \text{in}(\text{return p}) = \Phi_p(\text{in}(\text{call p})) \]
  \[ \text{out}(\text{return p}) = \text{in}(\text{return p}) \]
  AVOIDS PROPAGATION along exit-return edges!

• At entry p
  \[ \text{in}(\text{entry p}) = V \text{in}(\text{call p}) \]
  (propagates facts from all callers to callee)
Call String Approach

- A call string records outstanding calls in a path.

- E.g., call string \((c_1)(c_2)\) denotes that “we got there” on a path with outstanding calls at \(c_1\) and at \(c_2\).

\begin{align*}
1. \text{read } a, b \\
t = a \times b \\
2. \text{call } p \\
3. \text{return } p \\
4. t = a \times b \\
\text{print } t \\
5. \text{entry } p \\
6. \text{if } a == 0 \text{ then } a = a - 1 \\
7. \text{call } p \\
8. \text{return } p \\
9. \text{exit } p
\end{align*}
Call String Approach

- Tags solutions per program point with corresponding call string
- Multiple tagged solutions per program point $j$ in $p$:
  - Sharir and Pnueli Example:
  - We have $< \{ a*b \}, (c_1), < \{ \}, (c_1(c_2))$ at 6
  - Meaning: $a*b$ is available at 6 on paths with outstanding call string $c_1$, but it is not available on paths with outstanding call string $c_1c_2$
Call String Approach

- Apply original transfer functions point-wise

- Apply on elements of the original, i.e., “intraprocedural” dataflow lattice
  - \{ a*b \}, \{ a*b, a+b \}, {}, etc.

- Extend to handle call-entry and exit-return
  - At call-entry, simply append \( (c_i \)
  - At exit-return, propagate only if \( )_c_i \) matches!
Call String Approach

1. Extend in/out sets to sets of “tagged” lattice elements.
2. Apply orig. transfer funcs. point-wise.
3. Extend to handle call-entry, exit-return edges.

1. read a, b
   t = a*b

2. call p
   <{a*b},_>
   c1:

3. return p
   3. return p
   t = a*b
   print t

5. entry p
   <{a*b},(c1>

6. if a == 0 then
   a = a - 1

7. call p
   <{},(c1>
   c2:

8. return p
   t = a*b
   <{a*b},(c1>

9. exit p
Call String Approach

1. \( a = \&x \)
2. \( p = a \)
call id
3. return id
   \( b = \text{ret} \)
4. \( z = *b + *b \)
   \( c = \&y \)
5. \( p = c \)
call id
6. return id
   \( d = \text{ret} \)
7. entry id
   \( \text{ret} = p \)
8. exit id
Call String Approach

- At exit nodes, propagate only if open and close match!

\[
\langle \text{ret} \rightarrow x \rangle, \ (c_1), \\
\langle \text{ret} \rightarrow y \rangle, \ (c_2) \] at 9

Propagate \( \{\text{ret} \rightarrow y\} \) to 6, thus, \( \{d \rightarrow y\} \), because \( c_2 \) matches call string \( (c_2) \)

1. \( a = \& x \)
2. \( p = a \) call id
3. return id \( b = \text{ret} \)
4. \( z = *b + *b \) \( c = \& y \)
5. \( p = c \) call id
6. return id \( d = \text{ret} \)
7. entry id
8. ret = p
9. exit id
Call String Approach

- What is $S_{CS}(8)$?
  Union of $<p \rightarrow x, (c_1)>$ and $<p \rightarrow y, (c_2)>$ so $S_{CS}(8)$ is graph $\{ p \rightarrow x, p \rightarrow y \}$

- What is $S_{CS}(4)$?
- What is $S_{CS}(6)$? (out(6) more precisely)
Sharir and Pnueli, Key Result

- $S_{FA}(j)$ is the solution at $j$ computed by the functional approach
- $S_{CS}(j)$ is the solution at $j$ computed by the call string approach
- For (certain) distributive functions and finite lattices
  \[ S_{FA}(j) = S_{CS}(j) = \text{MORP}(j) \]

Caveats?
Sharir and Pnueli, Key Result

- Caveats
  - Summary functions $\Phi_p$ difficult to compute
  - With recursion, infinite call strings, $S_{CS}$ is infinite
  - Even for distributive functions and finite lattices, $S_{FA}$ and $S_{CS}$ cannot be computed (efficiently)

- Simple programming model
- Only distributive analysis
Key Points So Far

- ICFG
- Realizable paths
  - Use context-free grammar to describe
  - MORP
  - Goal of context-sensitive analysis is to filter out unrealizable paths, as much as possible
- Classical ideas
  - Functional approach
  - Call-string approach
Outline of Today’s Class

- Interprocedural control-flow graph (ICFG)
- Classical ideas in interprocedural analysis
- Context-sensitive analysis in practice
  - Notion of calling context
  - Call-string-based context sensitivity
  - Cloning-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

- Context-sensitive analysis in practice: ad-hoc adaptation of Sharir and Pnueli’s call string or functional approach

- 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
Notion of Calling Context

- **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
   \( b = \text{ret}_c1 \)

4. \( z = *b + *b \)
   \( c = \&y \)

5. \( p_{c2} = c \)
   call id_c2

6. return id_c2
   \( d = \text{ret}_c2 \)

7. entry id_c1

8. \( \text{ret}_c1 = p_{c1} \)

9. exit id_c1

10. entry id_c2

11. \( \text{ret}_c2 = p_{c2} \)

12. exit p_{c2}
Problems?

main:

... 

a = &x;

c1:  b = id(a);

z = *b + *b;

c = &y;

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?
Problems with k-CFA?

- 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 $m$
    $\{ o_i \} \subseteq \text{Pt}(x)$
  - Note: $\text{Pt}(x)$ denotes the points-to set of $x$
- Natural progression: RTA $\Rightarrow$ XTA $\Rightarrow$ 0-CFA $\Rightarrow$ PTA
Recall: PTA Constraints

\[ a_i : x = \text{new } A // o_i \quad \{ o_i \} \subseteq \text{Pt}(x) \]
\[ x = y \quad \text{Pt}(y) \nsubseteq \text{Pt}(x) \]
\[ x.f = y \quad \text{for each } o \text{ in } \text{Pt}(x). \text{Pt}(y) \subseteq \text{Pt}(o.f) \]
\[ x = y.f \quad \text{for each } o \text{ in } \text{Pt}(y). \text{Pt}(o.f) \subseteq \text{Pt}(x) \]

\[ c_i : x = y.m(z) \]
\[ \text{for each } o \text{ in } \text{Pt}(y) \]
\[ \text{let } m'(this,p,ret) = \text{resolve}(o,m) \text{ in} \]
\[ \{ o \} \subseteq \text{Pt}(this) \]
\[ \text{Pt}(z) \subseteq \text{Pt}(p) \quad \text{Pt}(ret) \subseteq \text{Pt}(x) \]
public class A {
    public static void main() {
        X x1 = new X();   // o
        A a1 = new B();   // o
        x1.f = a1;        // o.f points to o
        A a2 = x1.f;      // a2 points to o
        a2.m();
    }
}
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();

        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();
    }
}
Another PTA Example

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; }
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; }
Will Continue Next Time?
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:

PTA
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);
    }
}

Spring 21 CSCI 4450/6450, A Milanova
main() {
    Context theContext = new Context();

    c1: BoolExp or1 = new OrExp(new VarExp("X"), // or₁
                                new VarExp("Y"));
    c2: BoolExp or2 = new OrExp(new Constant(true), // or₂
                                new Constant(false));

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

Boolean Expression Hierarchy:
How About 1-CFA?
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: super(left, right); // call to constructor BinaryExp.<init>
}

...}
main() {
    Context theContext = new Context();

    // c1: Boolean expression 1
    BoolExp or1 = new OrExp(new VarExp("X"),
                            new VarExp("Y"));

    // c2: Boolean expression 2
    BoolExp or2 = new OrExp(new Constant(true),
                            new Constant(false));

    // c3: Evaluate expression 1
    boolean result1 = or1.evaluate(theContext);

    // c4: Evaluate expression 2
    boolean result2 = or2.evaluate(theContext);
}
What If We Changed Boolean Expression Hierarchy: 1-CFA?
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 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.

```java
A a = new A();  // o₁

// o₂
A a₁ = new A();
c₁: a₁.set_o₁(new X());

// o₃
A a₂ = new A();
c₂: a₂.set_o₁(new X());

// o₄
A a₃ = new A();
c₃: a₃.set_o₄(new Y());
```

- Again, flow-insensitive and context-sensitive, reaches our “ground truth”
Cloning-Based Context Sensitivity

class A {  <init>(X p) { this.f = p; } … } 
class B extends A { <init>(X p) { c1: super(p); } }  
  Note: super calls A.<init>(p) 
class C extends B { <init>(X p) { c2: super(p); } }  

c = new C; // o₁  
c3: c.<init>(new X()); // o₂  
c2 = new C; // o₃  
c4: c2.<init>(new X()); // o₄  

1-CFA?  
2-CFA?  
3-CFA?
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)
  - \( a.set(x) \) “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 \( a \times b \) is NOT available 😞

```
1. read a, b
   t = a*b
2. call p
3. return p
4. 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
```

\( c_1 \)
Key Points

- Context-sensitive analysis is difficult
- Different approaches
  - Call-string-based, also known as k-CFA
    - 2-CFA and 3-CFA
  - Cloning-based
  - Summary-based