Interprocedural Analysis and Context Sensitivity
Announcements

- Graded HW2
  - Fix issues before moving on to HW3
  - You can submit HW2 in Submitty

- One issue in virtual call **y.m()**
  - **static_type_of_y**
    - It is not `target.getDeclaringClass()`
    - It is `((RefType) args.get(0).getType()).getSootClass()` in `virtualCallStmt`

CSCI 4450/6450, A Milanova
So Far

- Four classical dataflow problems
  - **Intra**procedural
  - Flow-sensitive
- Class analysis: RTA, XTA, 0-CFA, and PTA
  - **Inter**procedural
  - Flow-insensitive and context-insensitive analysis
- Interprocedural analysis and context sensitivity
Outline of Today’s Class

- 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
Outline of Today’s Class

- Context-sensitive analysis in practice
  - Notion of calling context
  - Call-string-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
  - A **return** node and an **exit-return** edge
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 \( \text{KTA, O-CFA, PTA, RTA} \)
    - E.g., Andersen’s points-to analysis for C
int id(int p) {
    return p;
}

a = 5;
c1: b = id(a);
x = b*b;
c = 6;
c2: d = id(c);
int id(int p) {
    return p;
}

a = 5;
c1: b = id(a);
    x = b*b;
    c = 6;
c2: d = id(c);
Context-Insensitive Analysis

- Problem with context-insensitive analysis: propagates data along “unrealizable paths”

- Goal of context-sensitive analysis is to propagate data along “realizable paths”
int id(int p) {
    return p;
}

a = 5;
c1: b = id(a);
x = b*b;
c = 6;
c2: d = id(c);
Another Example (p3 from HW3)

class A {
    main() {
        B b = new B();
        b.m();
        A a = b.n();
        a.m();
    }
    A n() { return this; }
}

class B extends A {
    void m() {
        A a = new A();
        A a2 = a.n();
    }
}

O-CFA: a: {A, B3}
GT: a: {B3}

O-CFA propagates A to a.m() in main along the unrealizable orange path.
Another Example

int fib(int z, int u) {
    if (z<3) {
        return u+1; /* ret = u+1; */
    } else {
        c2: v = fib(z-1,u); /* auxiliary variable ret holds the return values */
        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. call
2. call
3. return
4. entry
5. z < 3
6. ret = u + 1
7. exit

fib:
8. call
9. return
10. call
11. return

T
F

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

- Context-free grammar!
- Grammar describes same-level path (SLP):

  \[ M ::= e \]
  \[ e \text{ denotes intra-procedural edge} \]
  \[ (c_i M)_{c_i} \text{ path from call to return} \]
  \[ M_1 M_2 \]

  - 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 \( \text{SLP}_{m,n} \) iff string along \( p \) is in language described by \( M \)
Realizable Paths (RP)

- What about paths with outstanding calls (calls that have not yet returned)?
- Another grammar:

\[
C ::= (c_i \mid M (c_i \mid C (c_i \mid C M
\]

- 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 Constant prop 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 \( \text{RP}_{1,7} \)? \( \text{YES} \)

Is \( p_2 = 1, 2, 4, 5, 8, 4, 5, 6, 7, 3 \) in \( \text{RP}_{1,3} \)? \( \text{No} \)

CSCI 4450/6450, A Milanova
Meet Over All Realizable Paths (MORP)

- \( \text{MORP}(n) = \bigvee f_{n_k} \circ f_{n_{k-1}} \circ \ldots \circ f_{n_2} \circ f_1(\text{init}) \)

  - \( p=(1,n_2\ldots n_k,n) \) is a path in 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 flow over realizable paths, as much as possible
Outline of Today’s Class

- 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
Sharir and Pnueli Example
(Available Expressions)

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
   else
   end if
   a = a - 1

7. call p

8. return p
   t = a*b

9. exit p

Q: Is a*b available at 4?
   GT: yes!
   CI: no!
   CS: yes!

unrealizable path:
1, 2, 5, 6, 7, 5, 6, 9, 3
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:
  - $\text{in(}\text{return } p\text{)} = \Phi_p(\text{in(}\text{call } p\text{)})$
  - thus, avoids propagation from callee along the exit $p$ --- $\text{return } p$ edge!
Phase 1:
Compute **summary transfer functions** $\Phi_p$ that capture effect of $p$. In example $\Phi_p$ is the **identity function**: nothing gets generated and nothing gets killed (simplifying a bit)
Functional Approach

Phase 2: Dataflow analysis:

1. \textbf{At return }p \textbf{in}(\text{return }p) = \Phi_p(\text{in}(\text{call }p))
2. \textbf{out}(\text{return }p) = \text{in}(\text{return }p)
   AVERTS PROPAGATION along exit-return edges!

3. \textbf{At entry }p \text{ in}(\text{entry }p) = V \text{ in}(\text{call }p)
   \text{(propagates facts from all callers to callee)}

1. read a, b
   \( t = a \times b \)

2. call p

3. return p

4. \( t = a \times b \)
   \text{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
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 \(c_2\).

\[
1. \text{read } a, b \\
t = a \cdot b \\
2. \text{call } p \\
3. \text{return } p \\
4. t = a \cdot b \\
\text{print } t \\
5. \text{entry } p \\
6. \text{if } a == 0 \text{ then} \\
a = a - 1 \\
7. \text{call } p \\
8. \text{return } p \\
9. \text{exit } p
\]
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 $c_1 c_2$
Call String Approach

- Apply original transfer functions point-wise on elements of the original, i.e., "intraprocedural" dataflow lattice
  - Elements: \{ a*b \}, \{ a*b, a+b \}, {}, etc.

- Extend to handle call-entry and exit-return
  - At call-entry, simply append \( \left( c_i \right) \)
  - At exit-return, propagate only if \( \right( c_i \) matches!
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
   \[ t = a\times b \]

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
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
  \[ S_{FA}(j) = S_{CS}(j) = MORP(j) \]
- 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 in general

- Simple programming model
- Only distributive analysis
Outline of Today’s Class

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

- We’ll continue next time

Reading
- Chapter 12.1-3 Dragon book
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
- 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