Interprocedural Analysis and Context Sensitivity
Announcements

- Quiz 3

- Quiz 1, Quiz 2, HW1, HW2 graded
  - Check Rainbow grades
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)
  - 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
Add procedure entry node and exit node
At each procedure call add
- A call node and a call-entry edge

2.call \longrightarrow 7.entry

- A return node and an exit-return edge

3.return \longleftarrow 9.exit
int* id(int* p) {
    return p;
}

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

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

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`
Unrealizable Paths

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

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

(c1)
(c2)
int fib(int z, int u) {
    if (z<3) {
        return u+1; /* ret = u+1; */
    } else {
        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. \( z = x \)
2. \( u = 0 \)
3. \( y = \text{ret} \)
4. \( \text{entry} \)
5. \( z < 3 \)
6. \( \text{ret} = u + 1 \)
7. \( \text{exit} \)

fib:
8. \( \text{call} \)
9. \( \text{return} \)
10. \( \text{call} \)
11. \( \text{return} \)

\[
\begin{align*}
\text{fib}(n) &= \begin{cases} 
1 & \text{if } n = 0 \\
2 & \text{if } n = 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1
\end{cases}
\end{align*}
\]
Realizable Paths (RP)

- Context-free grammar!
- Grammar describes same-level path (SLP):
  \[ M ::= e \quad \text{e denotes intra-procedural edge} \]
  \[ | (c_i M)_{c_i} \quad \text{path from call to return} \]
  \[ | M M \]
  - 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 \(M\)
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. 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
c1
8. ret = p
c2
9. exit id
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 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 \( p1 = 1,2,4,5,6,7 \) in \( \text{RP}_{1,7} \)?

Is \( p2 = 1,2,4,5,8,4,5,6,7,3 \) in \( \text{RP}_{1,3} \)?
Meet Over All Realizable Paths (MORP)

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

(\( \circ \) denotes function composition)
- Also called MVP (meet over all valid paths) or just MRP

- MORP(n) \( \leq \) 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
Classical 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
Sharir and Pnueli Example (Available Expressions)

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

2. call p

c1:

3. return p

4. \( t = a \times b \)
   print t

5. entry p

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

c2:

7. call p

8. return p
   \( t = a \times b \)

9. exit p

\( c1 \) \( c2 \)
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!
  - In a realizable path, a “kill” 6,7 is succeeded by call-entry 7,5, which must be balanced by exit-return 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 $p \rightarrow \text{return } p$ edge!
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)
Functional Approach

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)

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

CSCI 4450/6450, A Milanova
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 \(c_2\)
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 $< \{ \text{a*b} \}, (c_1 >, \ < \{ \}, (c_1(c_2 >$ at 6
    - Meaning: $\text{a*b}$ 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

- 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
   \[ t = a \times b \]

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

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. \( \text{ret} = p \)

9. exit id

\(<\{a \rightarrow x\}, _{c1} >\)

\(<\{b \rightarrow x\}, _{c1} >\)

\(<\{c \rightarrow y\}, _{c2} >\)

\(<\{d \rightarrow y\}, _{c2} >\)

\(<\{p \rightarrow x\}, (c_1) >\)

\(<\{p \rightarrow y\}, (c_2) >\)

\(<\{\text{ret} \rightarrow x\}, (c_1) >\)

\(<\{\text{ret} \rightarrow y\}, (c_2) >\)
Call String Approach

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

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

- What is $S_{\text{CS}}(4)$?
- What is $S_{\text{CS}}(6)$?
  (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
  - Context-free grammar describes realizable paths
  - MORP
  - Goal of context-sensitive analysis is to filter out unrealizable paths, as much as possible
- Classical ideas
  - Functional approach
  - Call-string approach