Announcements

- Quiz 2
- HW2?
  - Submitty page now open; tests with toy programs
  - You need a Java 7 rt.jar. Download one from the Oracle website if you don’t have one locally
  - On Windows, path separator is ; not :
  - Compile to .class, and give .class as input to Soot

Outline of Today’s Class

- ICFG
- Realizable paths, Meet-Over-all-Realizable-Paths (MORP) solution
  - Also denoted as MVP or MRP
- Classical results on interprocedural analysis
  - Call-string approach
  - Functional approach
- Context sensitive analysis in practice

Outline of Today’s Class

- Reading:
  - Sharir and Pnueli, “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.”
  - 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

Interprocedural Control Flow Graph (ICFG)

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

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

```java
a = &x;
c1: b = id(a);
z = *b + *b;
c = &y;
c2: d = id(c);
```
Context-Insensitive Analysis

- Add explicit assignments at call/return
  - E.g., \( x = \text{id}(y) \)
  - \( p = y \) models flow from actuals to formals
  - \( x = \text{ret} \) models flow from return to lhs
- Treat ICFG as one big CFG, and apply worklist algorithm
- Problem: merges data from different contexts
- Goal: track "realizable paths". **Context-sensitive** analysis tracks "realizable paths"

Infeasible 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
3. return id
4. z = *b + *b
5. p = c
6. return id
d = ret
```

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
3. return id
4. z = *b + *b
5. p = c
6. return id
d = ret
```

Another Example

```
int fib(int z, int u) {
  if (z<3) {
    return u+1;
    /* ret = u+1; */
  } else {
    v = fib(z-1,u);
    c2: v = fib(z-2,v)
    return fib(z-2,v)
  }
  ...
c1: y = fib(x,0);
```

```
What does fib compute? Here \( z \) and \( u \) are formal parameters; \( \text{ret} \) is the special variable holding the return value.
```

Realizable Paths (RP)

- Context-free grammar!
- Same-level path (SLP):
  - \( M ::= e \quad | \quad M \ | \quad (c_i M) \quad | \quad (c_i) \quad | \quad M M \)
  - An **intra**procedural edge is annotated with \( e \)
  - Call edge that originates at call site \( c_i \) is \( (c_i) \)
  - Corresponding return edge is \( (c_i) \)
  - A path \( p \) from \( m \) to \( n \) is in SLP if \( p \) is in language described by \( M \)
Realizable Paths (RP)

- Another grammar, describes paths with outstanding calls (i.e., calls not yet returned):
  \[ C ::= \{c_i \mid M \{c_i \mid C M \} \] 

- 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 \( \text{fib} \), 1,2,4,5,6,7 is in \( \text{RP} \) but 1,2,4,5,8,4,5,6,7,3 is not in \( \text{RP} \)

Meet Over All Realizable Paths (MORP)

- \( \text{MORP}(n) = V f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1(\text{init}) \) 
  \( p(n,\ldots,\text{init}) \) is a path in \( \text{RP}_{p(n)} \) 
  
\( \circ \) 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

Classic Results and Ideas

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981 
  - A finite lattice of dataflow facts 
  - Distributive transfer functions 
  - No local variables, and no parameter passing

Sharir and Pnueli Example (Available Expressions)

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 Example

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

- Expression \( a \times b \) is available at 4 if we consider only realizable paths 
  - Paths such as 1,2,5,6,7,5,6,9,3,4 is not realizable because the return edge 9,3 does not match the call edge 7,5

Functional Approach to Interprocedural Dataflow Analysis

- Operates on unchanged property space 
- Computes summary transfer functions \( \Phi_p \) that summarize effect of procedures \( p \) 
- 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 \( p \) edge! 
  - \( S_{\Phi_p}(j) \) is the solution at \( j \) computed by functional approach
Functional Approach

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

Phase 2:
Dataflow analysis:
- At return $p$ in $p$ = $\Phi_p$ in $p$ (AVOIDS PROPAGATION along exit-return edges!)
- At entry $p$ in $p$ = V in $p$ (propagates facts from all callers to callee)

Computing Summary Transfer Functions

- $\Phi_j$ is the summary of the effect of procedure $p$ from entry of $p$ up to node $j$

For certain well-behaved lattices and function spaces, we can solve these equations over functions, and compute summary transfer functions:

In general, not clear how to compute $\Phi$’s, even for finite lattices and general distributive function spaces.

Call String Approach to Interprocedural Dataflow Analysis

A call string records outstanding calls in 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 solutions per program point $j$ in $p$:
  Sharir and Pnueli Example:
  - We have $<\{a*b\}, <\{\}, \{c_1, c_2\}>$ at 6
  - Meaning: $a*b$ is available at 6 on paths with outstanding call string $c_2$, 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 original dataflow lattice elements
  - \( \{ a*b \}, \{ a*b, a+b \}, \emptyset \), etc.
- \( S_{CS}(j) \), the solution computed by the call-string approach, is the join over all call strings

Call String Approach

- At return nodes, propagate only matching call strings!
- \(< \text{ret} \rightarrow x \), \( c_1 \) >, \(< \text{ret} \rightarrow y \), \( c_2 \) > at 9

Sharir and Pnueli, Key Result

- For distributive functions and finite lattices
  \( S_{FA}(j) = S_{CS}(j) = \text{MORP}(j) \)

Caveats

- Summary functions difficult to compute...
- With recursion, we have infinite call strings...
- Therefore, even in the case of distributive functions and finite lattices, \( S_{FA} \) and \( S_{CS} \) cannot be (easily) computed