Announcements

- Quiz 1-2, HW1 now in Rainbow Grades
  - If you got points off, come by to get your paper
- HW2 due today?
- HW3: XTA Analysis
  - Will post the official assignment today
  - Submitty HW3 open tonight

Outline of Today’s Class

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

Reading

- Sharir and Pnueli’s “Two approaches to Interprocedural dataflow analysis”, 1981
- Dragon book, Chapter 12.1-3
- Thomas Reps, Susan Horwitz and Mooly Sagiv, “Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL’95

Classic Results and Ideas

- 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, and no parameter passing
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, \( a*b \) gets "killed" due to \( a = a - 1 \), and it is not recomputed

Functional Approach to Interprocedural Dataflow Analysis

- Operates on "intraprocedural" property space
- Computes summary transfer functions \( \Phi_p \) that summarize 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 \text{exit } p --- \text{return } p \text{ edge}
- \( S_{F_\Phi}(j) \) is the solution at \( j \) computed by functional approach

Functional Approach

Phase 1:
Compute a summary transfer function \( \Phi_p \) that captures effect of \( p \)
Assume \( \Phi_p \) is the identity function: nothing gets generated and nothing gets killed.

Phase 2:
Dataflow analysis:
- At \text{return } p
  - \( \text{in} (\text{return } p) = \Phi_p (\text{in}(\text{call } p)) \)
- at \text{entry } p
  - \( \text{in}(\text{entry } p) = V \text{ in}(\text{call } p) \)

Computing Summary Transfer Functions

- For certain lattices and function spaces, we can compute summary transfer functions
  - The IFDS framework we discuss today
- In general, not clear how to compute \( \Phi \)'s efficiently
- Ad-hoc approaches/approximation when computing \( \Phi \)'s for specific monotone function spaces (points-to analysis, taint analysis)
Call String Approach to Interprocedural Dataflow Analysis

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

1. read \(a, b\)
2. call \(p\)
3. return \(t = a*b\)
4. \(t = a*b\) print \(t\)
5. entry \(p\)
6. if \(a == 0\) then \(a = a - 1\)
7. call \(p\)
8. return \(p\)
9. 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 \(c_1 c_2\).

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.
4. \(t = a*b\) print \(t\)
5. entry \(p\)
6. if \(a == 0\) then \(a = a - 1\)
7. call \(p\)
8. return \(p\)
9. exit \(p\)

Call String Approach

- At exit nodes, propagate only matching call strings.

\(<\{a*b\}, c_1>, <\{\}, c_1 c_2>\) at 9.
Propagate \(\{ret \rightarrow y\}\) to 6.

Thus, \(\{d \rightarrow y\}\), because \(c_2\) matches call string \(c_1 c_2\).
Call String Approach

- What is $S_{CS}(8)$?
  Union of $<p \rightarrow x, \langle c_1 \rangle >$ and $<p \rightarrow y, \langle c_2 \rangle >$ 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
$S_{FA}(j) = S_{CS}(j) = MORP(j)$

Caveats?

Outline of Today’s Class

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity

- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

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

Caveats

- Summary functions 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)

Context-Sensitive Analysis In Practice

- Context-sensitive analysis in practice: ad-hoc variants of Sharir and Pnueli’s call string and functional approaches
- 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
Call String-Based Context Sensitivity

- **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 = ret_c1
4. z = *b + *b
   c = &y
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
```

Problems?

```
main: id:
...
  a = &x;
  c1: b = id(a);
  z = *b + *b;
  c = &y;
  c2: d = id(c);
  ...
  c3: return id_impl(p);
  ...
```

Strongly-Connected Components

- p forms a SCC.
  - Treat calls within p context-insensitively (thus concluding that a*b is not available at 4 \( \otimes \))
  - Analyze p per c1, other calling contexts if any

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
- Transfer functions (constraints) over syntax
  - E.g., Allocation $x = \text{new } A // o_i$
    - for each reachable method $m$
      - for each Allocation site $x = \text{new } A // o_i$ in $m$
        - $\{ o_i \} \subseteq \text{pts}(x)$
  - Note: $\text{pts}(x)$ denotes the points-to set of $x$
- Natural progression: RTA => XTA => 0-CFA => PTA

**PTA Example**

```java
c1: a = new A(); // o1
    x = new X(); // o2
    a.set(x); // o3
    x2 = new Y(); // o4
c2: a2.set(x2);

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

**Boolean Expression Hierarchy: PTA**

```java
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) {
        if (left.evaluate(c)) return true;
        if (right.evaluate(c)) return true;
        return false;
    }
}
```

**How About 1-CFA?**

```java
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);
}
```
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) {
        super(left, right); // call to constructor BinaryExp.<init>
    }
    ...
}

What If We Changed Boolean Expression Hierarchy? 1-CFA?
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), // or1
                                new Constant(false));
    c3: boolean result1 = or1.evaluate(theContext);
    c4: boolean result2 = or2.evaluate(theContext);
}

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
- It is more effective if we "cloned" method set per receiver object rather than per call site
  A a = new A(); // o1
  c1: a.set_o1(new X()); // o2
  c2: a.set_o1(new X()); // o3
  A a2 = new A(); // o4
  c3: a2.set_o4(new Y()); // o5
  Again, flow-insensitive and context-sensitive, reaches our "ground truth"

Cloning-Based Context Sensitivity
- Again, flow-insensitive and context-sensitive, reaches our "ground truth"
**Summary-based Context**

- 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

```
43
1. read a, b
2. call p
3. return p
4. t = a*b
5. entry p
6. if a == 0 then
   a = a - 1
7. call p
8. return p
9. exit p
```

**Summary-based Context**

- For a class of lattices and transfer functions one can represent functions, and compute summary transfer functions efficiently!