Interprocedural Analysis and Context Sensitivity

Announcements
- HW2?
- Submitty page still not open... Will test toy programs

Outline of Today's Class
- Overview of homework
- Catch-up: Class analysis
- XTA
- 0-CFA
- Points-to Analysis (PTA)

XTA
- \( R \) is the set of reachable methods
- \( S_m \) is the set of types that flow to method \( m \)
- \( S_f \) is the set of types that flow to field \( f \)

1. \( \{ \text{main} \} \subseteq R \)

2. for each method \( m \in R \) and each new site new \( C \) in \( m \)
   \( \{ C \} \subseteq S_m \)

3. for each method \( m \in R \), each virtual call \( y.n(z) \) in \( m \), each class \( C \) in \( \text{SubTypes}(\text{StaticType}(y)) \cap S_m \) and \( n' \), where \( n' = \text{resolve}(C,n) \)
   \( \{ n' \} \subseteq R \) \( \cap \) add \( n' \) to \( R \) if not already there
   \( \{ C \} \subseteq S_{n'} \) \( \cap \) add \( C \) to \( S_{n'} \) if not already there
   \( S_m \cap \text{SubTypes}(\text{StaticType}(p)) \subseteq S_{n'} \)
   \( S_{n'} \cap \text{SubTypes}(\text{StaticType}(\text{ret})) \subseteq S_m \)
   \( \text{if } (p \text{ denotes the parameter of } n', \text{ and } \text{ret denotes the return of } n') \)

4. for each method \( m \in R \), each field read \( x = y.f \) in \( m \)
   \( S_f \subseteq S_m \)

5. for each method \( m \in R \), each field write \( x.f = y \) in \( m \)
   \( S_m \cap \text{SubTypes}(\text{StaticType}(f)) \subseteq S_f \)
Practical Concerns

- Multiple parameters
- Direct calls
  - either static invoke calls or special invoke calls
- Array reads and writes!
- Static fields

See Tip and Palsberg for more

Example: RTA vs. XTA

```java
class A {
    static void n1() {
        A a1 = new B(); // S_n1 approximates locals in n1
        a1.m();
    }
}
class B {
    void m() {
        // S_n1 = { B }
    }
}
class C {
    void m() {
        // S_n2 = { C }
    }
}
```

Main Method

```java
public class AndExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;

    public AndExp(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);
    }
}
```

```
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);
    }
}
```

Boolean Expression Hierarchy: RTA vs. XTA

```
main() {
    Context theContext = new Context();
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp exp = new AndExp(
        new Constant(true), new OrExp(x, y));
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}
```

0-CFA

- 0-CFA stands for 0-level Control Flow Analysis, where “0-level” stands for context-insensitive analysis
  - Will see 1-CFA, 2-CFA, … k-CFA
- Improves on XTA by storing even more information about flow of class types
0-CFA
R is the set of reachable methods
S_v is the set of types that flow to variable v
S_f is the set of types that flow to field f

1. \{ main \} \subseteq R
2. for each method m \in R and each new site x = new C in m
   \{ C \} \subseteq S_x

3. for each method m \in R, each virtual call x = y.n(z) in m,
   each class C in S_y and n', where n' = resolve(C,n)
   \{ n' \} \subseteq R
   \{ C \} \subseteq S_{\text{mis}}
   S_y \cap \text{SubTypes(StaticType}(p)) \subseteq S_p
   S_{\text{ret}} \cap \text{SubTypes(StaticType}(x)) \subseteq S_x
   (this is the implicit parameter of n', p is the parameter of n', and ret is the return of n')

4. for each method m \in R, each field read x = y.f in m
   S_f \cap \text{SubTypes(StaticType}(x)) \subseteq S_x

5. for each method m \in R, each field write x.f = y in m
   S_f \cap \text{SubTypes(StaticType}(f)) \subseteq S_f

Example: XTA vs. 0-CFA

```java
public class A {
    public static void main() {
        A a1 = new B();
        a1.m();
        A a2 = new C();
        a2.m();
    }
}
```

Boolean Expression Hierarchy:

```
main() {
    Context theContext = new Context();
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp exp = new AndExp(
        new Constant(true),
        new OrExp(x, y)
    );
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}
```
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);
    }
}

Widely referred to as Andersen’s points-to analysis for Java

Improves on 0-CFA by storing information about objects, not classes

A a1 = new A(); // o1...
A a2 = new A(); // o2...

PTA

R is the set of reachable methods
pts(v) is the set of objects that v may point to
pts(o.f) is the set of objects that field f of object o may point to
1. { main } ⊆ R
2. for each method m ∈ R and each new site i: x = new C in m
   { o_i } ⊆ pts(x) // instead of C, we have o_i

3. for each method m ∈ R, each virtual call x = y.n(z) in m, each class o_i in pts(y)
   and n', where n' = resolve(class_of(o_i), n)
   { n' } ⊆ R
   { o_i } ⊆ pts(this)
   pts(z) ∩ SubTypes(StaticType(p)) ⊆ pts(p)
   pts(ret) ∩ SubTypes(StaticType(x)) ⊆ pts(x)
   (this is the implicit parameter of n', p is the parameter of n', and ret is the return of n')

4. for each method m ∈ R, each field read x = y.f in m
   for each object o ∈ pts(y)
   pts(o.f) ∩ SubTypes(StaticType(x)) ⊆ pts(x)

5. for each method m ∈ R, each field write x.f = y in m
   for each object o ∈ pts(x)
   pts(y) ∩ SubTypes(StaticType(f)) ⊆ pts(o.f)

6. for each method m ∈ R, each assign stmt x = y in m
   pts(y) ∩ SubTypes(StaticType(x)) ⊆ pts(x)
Example: 0-CFA vs. PTA

```java
public class A {
    public static void main() {
        X x1 = new X(); // o1
        A a1 = new B(); // o2
        x1.f = a1; // o1, f points to o2
        A a2 = x1.f; // a2 points to o2
        a2.m();

        X x2 = new X(); // o3
        A a3 = new C(); // o4
        x2.f = a3; // o3, f points to o4 equiv. pts(o3, f) = { o4 }
        A a4 = x2.f; // a4 points to o4
        a4.m();
    }
}
```

The Big Picture

- All fit into our monotone dataflow framework!
- Flow-insensitive, context-insensitive
  - Least solution of $S = f(S) \cup S$
- Algorithms differ mainly in "size" of $S$
  - RTA: only 2 kinds of statements; Lattice?
  - XTA: expands to all statements; Lattice?
  - 0-CFA: all statements; Lattice?
  - PTA (Points-to analysis): all statements; Lattice elements are points-to graphs

Outline of Today's Class

- Interprocedural Control Flow Graph (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
Interprocedural Control Flow Graph (ICFG)

```c
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/return
  - E.g., `x = id(y)`
  - `p = y` models flow from actuals to formals
  - `x = 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

```c
int* id(int* p) {
    return p;
}  
a = &x;  
c1: b = id(a);  
    z = *b + *b;  
    c = &y;  
c2: d = id(c);  
```

Realizable Paths

```c
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;  
    } 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

```c
main:  
1.  
   4.entry  
   F  
2. call  
   5.z=5  
   T  
3.return  
   y=ret  
5.return=u+1  
6. return=ret  
```

Spring 19 CSCI 4450/6450, A Milanova
Realizable Paths (RP)

- **Context-free grammar!**
- **Same-level (balanced) path (SLP):**
  \[ M ::= e | (c_i M) c_i \]
  - \( e \) denotes an intra-procedural edge
  - \( (c_i M) c_i \) captures path from call to return
  - 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 \( \text{SLP}_{m,n} \) if the string along \( p \) is in the language described by \( M \).

Another grammar, describes paths with outstanding calls (i.e., calls not yet returned):

\[ C ::= (c_i | M | M C) \]

- A path from entry node 1 to node \( n \) is in \( \text{RP}_{1,n} \) if 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 \( 1, 2, 4, 5, 6, 7, 3 \) is not in \( \text{RP} \).

Meet Over All Realizable Paths (MORP)

\[ \text{MORP}(n) = V f_1^n \circ f_2 \circ \ldots \circ f_n(\text{init}) \]

- \( f_i \) 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

Sharir and Pnueli Example (Available Expressions)

- **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

Classical Results and Ideas
Sharir and Pnueli Example

- Expression $a \cdot b$ is NOT available at 4 if we consider all paths
  - E.g., along 1, 2, 5, 6, 7, 5, 6, 9, 3, 4 $a \cdot b$ gets "killed" due to $a = a - 1$, and it is not recomputed
- Expression $a \cdot b$ is available at 4 if we consider only realizable paths
  - Path 1, 2, 5, 6, 7, 5, 6, 9, 3, 4 gets "killed" due to $a = a - 1$, and it is not recomputed

Functional Approach to Interprocedural Dataflow Analysis

- Operates on unchanged 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 exit $p \rightarrow \text{return } p$ edge!
- $S_{FSA}(j)$ is the solution at $j$ computed by functional approach

Functional Approach

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

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

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

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 $c_1: c_2$ 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 in the set \{p\}.
  - Sharir and Pnueli Example:
    - We have \{a*b\}, \{c1\} at 6.
  - Meaning: \{a*b\} is available at 6 on paths with outstanding call string \{c1\}, but it is not available on paths with outstanding call string \{c1, c2\}.

At exit nodes, propagate only matching call strings.
\{ret -> x\}, \{c1\}, \{ret -> y\}, \{c2\} at 9.
Propagate \{ret -> y\} to 6, thus \{d -> y\}, because \{d -> y\} matches call string \{c2\}.

Apply original transfer functions point-wise.
Apply on original dataflow lattice elements.
- \{a*b\}, \{a*b, a+b\}, {}, etc.

Graph:
- \{p -> x\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\}.
- \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\}.

What is \(S_{CS}(8)\)?
Union of \{p -> x\} and \{c1\}.
\{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\} \rightarrow \{p -> y\}.
- \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> y\} \rightarrow \{d -> 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?

- 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)