Program Analysis Frameworks and Dataflow Analysis: Catch-up

Announcements
- HW2 out
- Submitty page is up
- Post questions on forum
  - Setup
  - Starter code, generic analysis framework and fixpoint iteration algorithm
  - Soot

Outline of Today’s Class
- Program analysis frameworks: Soot & Ghidra
- Catch-up: Points-to analysis for C
- Catch-up: Class analysis
  - Class Hierarchy Analysis (CHA)
  - Rapid Type Analysis (RTA)
- XTA, 0-CFA, and PTA (if we have time)

Overview of Soot

Advantages of Jimple and Soot
- Jimple
  - Typed local variables
  - 16(!) simple 3-address statements (1 operator per statement). Bridges gap between analysis abstraction and analysis implementation
- Soot provides
  - Intraprocedural dataflow analysis framework
  - Points-to analysis
  - Context-sensitive analysis framework
  - Android taint analysis

Jimple
- Run soot: java soot.Main –jimple A (need paths)
  - public class A extends java.lang.Object
    - public void <init>() {
      - A a = new A();
      - a.m();
    }
    - public void m() {
      - }
    - }
    - return;
    - }
    - (continues on next slide…)
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
        ...
    }
}

public void m() {
    A r0;  
    r0 := @this: A;
    return;
}

Jimple:

Java:

public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
        ...
    }
}

public void m() {
    A r0;  
    r0 := @this: A;
    return;
}

Soot Abstractions. Look up API!

- Abstracts program constructs
- Some basic Soot classes and interfaces
  - SootClass
  - SootMethod
    - SootMethod sm; sm.isMain(), sm.isStatic(), etc.
  - Local
    - Local l; ... l.getType()
  - InstanceInvokeExpr
    - Represents an instance (as opposed to static) invoke expression
    - InstanceInvokeExpr iie; ... receiver = iie.getBase();

4 Kinds of Calls

- Constructor/Super Call:
  - A a = new A();
  - $r1 = new A;
  - specialinvoke $r1.<A: void <init>()>();
- Virtual Call:
  - a.m();
  - virtualinvoke r2.<A: void m()>();
- Static Call:
  - sm();
  - staticinvoke <A: void sm()>();
- Interface Call:
  - x.m();
  - interfaceinvoke r0.<pack2.X: void m()>();

1. We should not need to worry about dynamicInvoke. (Soot does support it.)

An Overview of Homework

- Syntax
  - Assignment stmt: x = y
  - Field read stmt: x = y.f
  - Field write stmt: x.f = y
  - Array read stmt: x = y[i]
  - Array write stmt: x[i] = y
  - Allocation stmt: x = new A;
  - Direct call: x = sm(args) or x = y.m(args)
  - Virtual call: x = y.m(args)

For RTA, we only care about the last 3
Outline of Today’s Class

- Program analysis frameworks: Soot & Ghidra
- Catch-up: Points-to analysis for C
- Catch-up: Class analysis
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- XTA, 0-CFA, and PTA (if we have time)

Points-to Analysis

- Problem statement: What memory locations may a pointer variable point to?
- Many applications!
  - Enables compiler optimizations
  - Static debugging tools, static taint analysis tools

Points-to Analysis: Example

Example 1:

```c
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```

Example 2:

```c
int a, b = 15;
int *p1, *p2;
int **p3;
p3 = &p1;
p1 = &a;
p2 = *p3;
*p2 = b;
```

Points-to Analysis: Transfer Functions

1. \( f_{\text{addr}} \): “kill” all points-to edges from \( p \) to \( q \)
2. \( f_{\text{prop}} \): “kill” all points-to edges from \( p \); “generate” new points-to edges from \( p \) to every \( x \), such that \( q \) points to \( x \) in incoming points-to graph \( j \)
3. \( f_{\text{ind}} \): “kill” all points to edges from \( p \); “generate” new points-to edges from \( p \) to every \( x \), s.t. there is \( y \) where \( q \) points to \( y \), and \( y \) points to \( x \) in \( j \)
4. \( f_{\text{dow}} \): Do not kill! Can you think of a reason why?
   - “Generate” new points-to edges from every \( y \) to every \( x \), such that \( p \) points to \( y \) and \( q \) points to \( x \)

Points-to Analysis: Property Space

- Lattice \( L, \leq \)
  - Lattice of the subsets over all edges \( p \to q \) where \( p \) and \( q \) are program variables
  - … or in simpler terms, lattice elements are points-to graphs, e.g.,
    - \( p_3 \)
    - \( V \) is points-to graph union
    - \( 0 \) of \( L \) is empty graph
    - \( 1 \) of \( L \) is complete graph
Points-to Analysis: Examples

Example 1:

\[ p_1 = \&a \]
\[ p_2 = p_1 \]
\[ *p_2 = 1 \]

Example 2:

\[ p_3 = \&p_1 \]
\[ p_1 = \&a \]
\[ \ldots \]
\[ q = p_3 \]
\[ r = *q \]
\[ p_1 = \&b \]

Points-to Analysis is Monotone

To argue monotonicity we must show that if \( Pt_1 \leq Pt_2 \), then \( f(Pt_1) \leq f(Pt_2) \) for each transfer function \( f \).

Points-to Analysis is Not Distributive

\[ p = \&x; \quad q = \&y; \]
\[ p = \&z; \quad q = \&w; \]
\[ *p = q \]

MFP vs. MOP for Points-to

Andersen’s Points-to Analysis

- Commonly attributed to Lars Andersen [1994]
- “Andersen’s points-to analysis for C”
- More approximation than our earlier formulation: don’t ever “kill”; maintain a single points-to graph for all program points
- Flow-insensitive, context-insensitive analysis
- Formulated in terms of subset constraints
- Solvable by a version of the fixpoint iteration

… but it is not distributive!

- Because of updates!
Andersen’s Points-to Analysis

$\text{pts}(p)$ denotes the points-to set of $p$

1. $p = \&a \{ a \} \in \text{pts}(p)$
2. $p = q \Rightarrow \text{pts}(q) \subseteq \text{pts}(p)$
3. $p = *q$ for each $x$ in $\text{pts}(q)$. $\text{pts}(x) \subseteq \text{pts}(p)$
4. $*p = q$ for each $x$ in $\text{pts}(p)$. $\text{pts}(q) \subseteq \text{pts}(x)$

Use worklist-like algorithm to compute least solution of these constraints

---

Andersen’s Points-to Analysis: Examples

Example 1:

```
p1 = &a
p2 = p1
*p2 = 1
```

Example 2:

```
p3 = &p1
p1 = &a
...
q = p3
r = *q
p1 = &b
```
Example

public class A {
    public static void main() {
        A a;
        D d = new D();
        E e = new E();
        if (...) a = d; else a = e;
        a.m();
    }
}

public class B extends A {
    public void foo() {
        G g = new G();
    }
}

RTA

R is the set of reachable methods
I is the set of instantiated types

1. \{ main \} \subseteq R // initialize R with main
2. for each method m in R and each new site new C in m
   \{ C \} \subseteq I

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. \{ main \} \subseteq R
2. for each method m in R and each new site new C in m
   \{ C \} \subseteq S_m // add C to S_m if not already there

XTA Analysis Family

- Due to Tip and Palsberg
  - Frank Tip and Jens Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms", OOPSLA '00
- Generalizes RTA
- Improves on RTA by storing more precise information about flow of class types
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
   \( S_f \cap \text{SubTypes}(\text{StaticType}(f)) \subseteq S_r \)

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
public class A {
    public static void main() {
        n1();
        n2();
    }
    static void n1() {
        A a1 = new B();
        a1.m();
    }
    static void n2() {
        A a2 = new C();
        a2.m();
    }
}
```

Boolean Expression Hierarchy: RTA vs. XTA vs. “Ground Truth”
```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);
    }
}
```

```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) {
        private BoolExp l = this.left;
        private BoolExp r = this.right;
        return l.evaluate(c) || r.evaluate(c);
    }
}
```

```java
public class 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);
}
```
Described in Tip and Palsberg’s paper

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 next time.

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 \} ⊆ 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’ = \( \text{resolve}(C,n) \)

\( \{ n' \} \subseteq R \)

\( \{ C \} \subseteq S_{\text{this}} \)  // this (impl. param) of n’

\( S_y \cap \text{SubTypes}(\text{StaticType}(p)) \subseteq S_y \)

\( S_f \cap \text{SubTypes}(\text{StaticType}(x)) \subseteq S_f \)

(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

\( \text{SubTypes}(\text{StaticType}(x)) \subseteq S_x \)

5. for each method m in R,
    each field write \( x.f = y \) in m

\( \text{SubTypes}(\text{StaticType}(f)) \subseteq S_f \)

**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: XTA vs. 0-CFA**

```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: XTA vs. 0-CFA

- 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

3. for each method m in R, each virtual call x = y.n(z) in m, each class o in Pt(y)
   and n', where n' = resolve(class_of(o),n) 
   { n' } ⊆ R
   { o } ⊆ Pt(this)
   Pt(z) ∩ SubTypes(StaticType(p)) ⊆ Pt(p)
   Pt(ret) ∩ SubTypes(StaticType(x)) ⊆ Pt(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
   for each object o ∈ Pt(y)
   Pt(o.f) ∩ SubTypes(StaticType(x)) ⊆ Pt(x)
   Pt(y) ∩ SubTypes(StaticType(f)) ⊆ Pt(o.f)

Spring 20 CSCI 4450/6450, A Milanova

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);
}

Spring 20 CSCI 4450/6450, A Milanova
The Big Picture

- All fit into the monotone dataflow framework!
- Flow-insensitive, context-insensitive
  - Least solution of $S = f(S) \lor S$
- Differ (most importantly) 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

Next class

- Quiz 2 on points-to analysis and RTA/XTA
- Interprocedural Analysis
- Context sensitivity

Example: 0-CFA vs. PTA

```java
class A {
    public static void main() {
        X x1 = new X(); // o_1
        A a1 = new B(); // o_2
        x1.f = a1; // o_4, f points to o_2
        A a2 = x1.f; // a2 points to o_2
        a2.m();

        X x2 = new X(); // o_3
        A a3 = new C(); // o_4
        x2.f = a3; // o_5, f points to o_4
        A a4 = x2.f; // a4 points to o_4
        a4.m();
    }
}
```