Class Analysis: Catch-up

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

Outline of Today’s Class
- Catch-up: Points-to analysis for C
- Catch-up: Class analysis
- Class Hierarchy Analysis (CHA)
- Rapid Type Analysis (RTA)
- XTA
- 0-CFA, and PTA
- Quiz 2

Points-to Analysis
- Problem statement: What memory locations may a pointer variable point to?
- Assume the following 4 simple statements
  1. address taken \( p = \&q \)
  2. propagation \( p = q \)
  3. indirect read \( p = *q \)
  4. indirect write (update) \( *p = q \)

Points-to Analysis: Transfer Functions
1. \( f_{p=\&q} \): “kill” all points-to edges from \( p \), and “generate” a new points-to edge from \( p \) to \( q \)
2. \( f_{p=q} \): “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 \( \text{in}(j) \)
3. \( f_{p=*q} \): “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 \( \text{in}(j) \)
4. \( f_{*p=q} \): 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 \( p \rightarrow q \) where \( p \) and \( q \) are program variables
  - … or in simpler terms, lattice elements are points-to graphs, e.g.,
    \[
    \begin{array}{c}
    p3 \\
    \downarrow \\
    p1 \\
    \hline
    p2 \\
    \hline
    a \\
    \end{array}
    \]
  - \( V \) is points-to graph union
  - \( 0 \) of \( L \) is empty graph
  - \( 1 \) of \( L \) is complete graph
Points-to Analysis is Monotone

To argue monotonicity we must show that if $P_t_1 \leq P_t_2$ then $f(P_t_1) \leq f(P_t_2)$ for each transfer function $f$

1. $P_t_1 \leq P_t_2$ then $f(p\&q)(P_t_1) \leq f(p\&q)(P_t_2)$
2. $P_t_1 \leq P_t_2$ then $f(p\&q)(P_t_1) \leq f(p\&q)(P_t_2)$
3. $P_t_1 \leq P_t_2$ then $f(p\&q)(P_t_1) \leq f(p\&q)(P_t_2)$
4. $P_t_1 \leq P_t_2$ then $f(p\&q)(P_t_1) \leq f(p\&q)(P_t_2)$

Points-to Analysis is Not Distributive

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!

MFP

MOP
Andersen’s Points-to Analysis

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

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

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:

- $p3 = \&p1$
- $p1 = \&a$
- $\ldots$
- $q = p3$
- $r = *q$
- $p1 = \&b$

Outline of Today’s Class

- Catch-up: Points-to analysis for C
- Catch-up: Class analysis
  - Class Hierarchy Analysis (CHA)
  - Rapid Type Analysis (RTA)
  - The XTA analysis family
  - 0-CFA
  - Points-to Analysis (PTA)

Class Analysis

- Problem statement: What are the classes of objects that a (Java) reference variable may refer to?
- Applications
  - Call graph construction
  - Virtual call resolution

Class Hierarchy Analysis (CHA)

- In Java, if a reference variable $r$ has type $A$, $r$ can refer only to objects that are concrete subclasses of $A$. Denoted by $\text{SubTypes}(A)$
  - Note: refers to Java subtype, not true subtype
  - Note: $\text{SubTypes}(A)$ notation due to Tip and Palsberg (OOPSLA ’00)
  - At virtual call site $r.m()$, we can find what methods may be called based on the hierarchy information

Rapid Type Analysis (RTA)

- Due to Bacon and Sweeney
  - David Bacon and Peter Sweeney, “Fast Static Analysis of C++ Virtual Function Calls”, OOPSLA ’96
- Improves on CHA
- Expands calls only if it has seen an instantiated object of the appropriate type
Example

```java
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();
    }
}
```

```java
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 \} ∈ R // initialize R with main

2. for each method m in R and each new site new C in m
   \{ C \} ⊆ I

Example RTA starts at main. Records that D and E are instantiated.
All call a.m() looks at all CRA targets. Expands only into target C.m()
Never reaches B.foo(), never records G as being instantiated.

RTA 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

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 \} ⊆ R

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

XTA

3. for each method m in R, each virtual call y.n(z) in m, each class C in \text{SubTypes}(\text{StaticType}(y)) ∩ S_m and n', where n' = resolve(C,n)
   \{ n' \} ⊆ R // add target n' to R, if not already there
   \{ C \} ⊆ S_f // add C to S_f, if not already there
   S_m ∩ \text{SubTypes}(\text{StaticType}(p)) ⊆ S_{n'}

(p denotes the parameter of n', and 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
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);
    }
}
```

---

**Boolean Expression Hierarchy:**

**RTA vs. XTA vs. “Ground Truth”**

```java
public class OrExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;
    private AndExp or = new AndExp(x, y);
    Context theContext = new Context();
    boolean result = exp.evaluate(theContext);
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
```
0-CFA

- 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

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_v

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_{this} // this (impl. param) of n'
   S_f \cap SubTypes(StaticType(p)) \subseteq S_{o}
   S_{ref} \cap SubTypes(StaticType(x)) \subseteq S_{r}
   (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 SubTypes(StaticType(x)) \subseteq S_{r}

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

Example: XTA vs. 0-CFA

```
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**

```
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);
    }
    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);
    }
}

main() {
    Context theContext = new Context();
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp tr = new Constant(true);
    BoolExp or = new OrExp(x, y);
    BoolExp exp = new AndExp(tr, or);

    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}

PTA

- 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
Pt(v) is the set of objects that v may point to
Pt(o.f) is the set of objects that field f of object o may point to
1. \{ main \} ⊆ R

2. for each method m in R and each new site i: x = new C in m
   \{ o_i \} ⊆ Pt(x) // instead of C, we have o_i

PTA

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

PTA

4. for each method m in R, each field read x = y.f in m
   for each object o in Pt(y)
   Pt(o.f) \cap SubTypes(StaticType(x)) ⊆ Pt(x)

5. for each method m in R, each field write x.f = y in m
   for each object o in Pt(x)
   Pt(y) \cap SubTypes(StaticType(f)) ⊆ Pt(o.f)
Example: 0-CFA vs. PTA

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
        a2.m();
        X x2 = new X();    // o₃
        A a3 = new C();   // o₄
        x2.f = a3;  // o₃.f points to o₄
        A a4 = x2.f; // a₄ points to o₄
        a4.m();
    }
}

The Big Picture

- All fit into the monotone dataflow framework!
- Flow-insensitive, context-insensitive
  - Least solution of $S = f(S) \cup V$ 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

The Big Picture

RTA:
Types: A B C D

XTA: $S_{m1} \ldots S_{mk} \ldots S_{mx}$

0-CFA: $v_1, v_2, \ldots, v_n$

PTA: $v_1, v_2, \ldots, v_n$

Next class

- Interprocedural Analysis
- Context sensitivity