Dataflow Analysis: Catch-up

Announcements
- HW2 out
- Should have Submitty page up today
- Post questions on form
  - Setup
  - Starter code, generic analysis framework and fixpoint iteration algorithm
  - Soot
- Quiz 2 at end of class

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
- Points-to Analysis (PTA)

Outline of Today's Class
- Reading
  - Frank Tip and Jens Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms", OOPSLA '00

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) $f_{\text{p}=&\text{q}}$: “kill” all points-to edges from $p$, and “generate” a new points-to edge from $p$ to $q$

(2) $f_{\text{p}=\text{q}}$: “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 $i(j)$

(3) $f_{\text{p}=^{*}\text{q}}$: “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 $i(j)$

(4) $f_{\text{p}=^{*}\text{q}}$: 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 $x$

Example 1:
$p1 = &a$
$p2 = p1$
*p2 = 1

Example 2:
$p3 = &p1$
$p1 = &a$
$q = p3$
*r = *q$
$p1 = &b$

Points-to Analysis is Monotone

To argue monotonicity we must show that if $P_{t1}$ is $\leq$ (subset of) $P_{t2}$, then $f(P_{t1}) \leq f(P_{t2})$ for each transfer function $f$

(1) $P_{t1} \leq P_{t2}$ then $f_{\text{p}=&\text{q}} (P_{t1}) \leq f_{\text{p}=&\text{q}} (P_{t2})$

(2) $P_{t1} \leq P_{t2}$ then $f_{\text{p}=\text{q}} (P_{t1}) \leq f_{\text{p}=\text{q}} (P_{t2})$

(3) $P_{t1} \leq P_{t2}$ then $f_{\text{p}=^{*}\text{q}} (P_{t1}) \leq f_{\text{p}=^{*}\text{q}} (P_{t2})$

(4) $P_{t1} \leq P_{t2}$ then $f_{\text{p}=^{*}\text{q}} (P_{t1}) \leq f_{\text{p}=^{*}\text{q}} (P_{t2})$

Points-to Analysis is Not Distributive

What for “$*p = q$” does: Adds edges from each variable that $p$ points to ($x$ and $z$), to each variable that $q$ points to ($y$ and $w$). Result is 4 new edges: from $x$ to $y$ and to $w$ and from $z$ to $y$ and to $w$. Result in $P_{t4} = f_{\text{p}=&\text{q}} (P_{t1} \lor P_{t1})$.

MFP vs. MOP for Points-to

1. $\text{if}(n>0)$

2. $\text{p} = \text{q} \lor \text{x} ; \text{q} = \text{y} ;$

3. $f_{\text{p}=&\text{q}} (P_{t3})$

4. $f_{\text{p}=\text{q}} (P_{t3})$

5. $f_{\text{p}=^{*}\text{q}} (P_{t4})$

6. $f_{\text{p}=^{*}\text{q}} (P_{t4})$

7. $f_{\text{p}=^{*}\text{q}} (P_{t4})$

8. $f_{\text{p}=^{*}\text{q}} (P_{t4})$

9. $f_{\text{p}=^{*}\text{q}} (P_{t4})$

10. $f_{\text{p}=^{*}\text{q}} (P_{t4})$
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

\[ \begin{align*}
\text{pts}(p) & \text{ denotes the points-to set of } p \\
1) & p = \&a \quad \{ a \} \subseteq \text{pts}(p) \\
2) & p = q \quad \text{pts}(q) \subseteq \text{pts}(p) \\
3) & p = \ast q \quad \text{for each } x \in \text{pts}(q), \text{pts}(x) \subseteq \text{pts}(p) \\
4) & \ast p = q \quad \text{for each } x \in \text{pts}(p), \text{pts}(q) \subseteq \text{pts}(x)
\end{align*} \]

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

Andersen’s Points-to Analysis: Examples

**Example 1:**
\[
\begin{align*}
p1 &= \&a \\
p2 &= p1 \\
P2 &= 1
\end{align*}
\]

**Example 2:**
\[
\begin{align*}
p3 &= \&p1 \\
p1 &= \&a \\
q &= p3 \\
r &= \ast q \\
p1 &= \&b
\end{align*}
\]

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)

- Attributed to Dean, Grove and Chambers:
  - Jeff Dean, David Grove, and Craig Chambers, “Optimization of OO Programs Using Static Class Hierarchy Analysis”, ECOOP’95
- Simplest way of inferring information about reference variables, simply look at class hierarchy!
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

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;
    }
}

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

RTA

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

RTA starts at \( \text{main} \).
Records that \( D \) and \( E \) are instantiated.
At call \( a.m() \) looks at all CHA targets.
Expands only into target \( C.m() \).
Never reaches \( B.foo() \), never records \( G \) as being instantiated.

XTA Analysis Family

- Due to Tip and Palsberg
  - Frank Tip and Jens Palsberg, “Scalable Propagation-Based Call Graph Construction Algorithms”, OOPSLA ‘00

Generalizes RTA

Implements 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. $\{ \text{main} \} \subseteq R$
2. for each method $m \in R$ and each new site $\text{new } C$ 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 $n' = \text{resolve}(C,n)$
   $\{ n' \} \subseteq R$ // add $n'$ to $R$ if not already there
   $\{ C \} \subseteq S_n'$ // add $C$ to $S_n'$ if not already there
   $S_m \cap \text{SubTypes}(\text{StaticType}(p)) \subseteq S_n'$
   $S_n' \cap \text{SubTypes}(\text{StaticType}(\text{ret})) \subseteq S_m$
   ($p$ denotes the parameter of $n'$, 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
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);
    }
}
```
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 exp = new AndExp(
        new Constant(true), new OrExp(x, y) );
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}