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

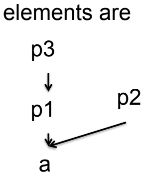
Points-to Analysis

- 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: Property Space

- Lattice **L,≤**
  - Lattice of the subsets over all edges **p → q** where **p** and **q** are program variables
  - … or in simpler terms, lattice elements are points-to graphs, e.g.,



  - **V** is points-to graph union
  - **0** of L is empty graph
  - **1** of L is complete graph

Points-to Analysis: Transfer Functions

(1) $\mathbf{f_{p=\&q}}$: "kill" all points-to edges from **p**, and "generate" a new points-to edge from **p** to **q**

(2) $\mathbf{f_{p=q}}$: "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 in(j)

(3) $\mathbf{f_{p=*q}}$: "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 in(j)

(4) $\mathbf{f_{*p=q}}$: 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: Examples

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

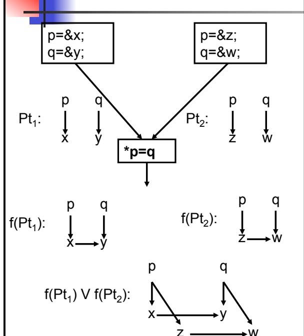
- To argue monotonicity we must show that if $\mathbf{Pt_1}$ is **≤** (subset of) $\mathbf{Pt_2}$, then $\mathbf{f(Pt_1)} \leq \mathbf{f(Pt_2)}$ for each transfer function **f**

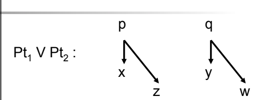
(1) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=\&q}}(Pt_1) \leq \mathbf{f_{p=\&q}}(Pt_2)$
(2) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=q}}(Pt_1) \leq \mathbf{f_{p=q}}(Pt_2)$
(3) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=*q}}(Pt_1) \leq \mathbf{f_{p=*q}}(Pt_2)$
(4) $Pt_1 \leq Pt_2$ then $\mathbf{f_{*p=q}}(Pt_1) \leq \mathbf{f_{*p=q}}(Pt_2)$

… but it is not distributive!

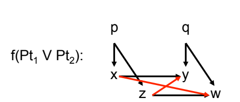
- Because of updates!

Points-to Analysis is Not Distributive

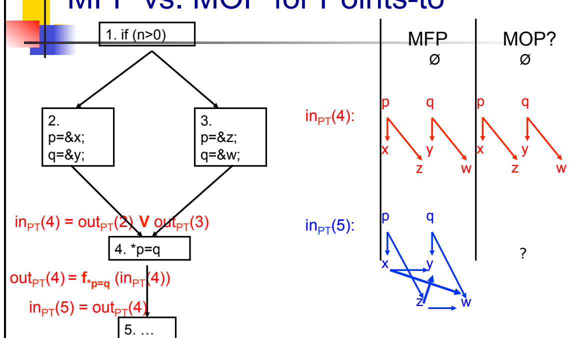




What **f** for ***p = q** does: Adds edges from each variable that **p** points to (**x** and **z**), to each variable that **q** points to (**y** and **w**). Result is 4 new edges:
from **x** to **y** and to **w**
and from **z** to **y** and to **w**



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

Andersen's Points-to Analysis

pts(p) denotes the points-to set of **p**

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


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!

Class Hierarchy Analysis (CHA)

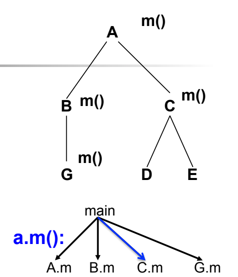
- In Java, if a reference variable **r** has type **A**, **r** can refer only to objects that are concrete subclasses of **A.** Denoted by **SubTypes(A)**
  - Note: refers to Java subtype, not true subtype
  - Note: **SubTypes(A)** notation due to Tip and Palsberg (OOPSLA'00)
- At virtual call site **r.m(),** 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

```
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 starts at **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.

RTA

R is the set of reachable methods

I is the set of instantiated types

1. **{** main **}** $\sqsubseteq$ **R** // initialize **R** with **main**

2. for each method **m** $\in$ **R** and each new site **new C** in **m**

   { C } $\sqsubseteq$ **I**

RTA

3. for each method **m** $\in$ **R**,
each virtual call **y.n(z)** in **m**,
each class **C** in **SubTypes(StaticType(y))** $\cap$ **I,**
and **n'**, where **n'** = **resolve(C,n)**

   { n' } $\sqsubseteq$ **R** // add target **n'** to **R**, 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

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** ∈ **R** and
   each new site **new C** in **m**
   { C } ⊑ S_m

XTA

3. for each method **m** ∈ **R**,
each virtual call **y.n(z)** in **m**,
each class **C** in **SubTypes(StaticType(y))**∩S_m
and **n'**, where **n'** = **resolve(C,n)**

   { n' } ⊑ **R** // add **n'** to **R** if not already there
   { C } ⊑ $S_{n'}$ // add **C** to $S_{n'}$ if not already there
   S_m ∩ **SubTypes(StaticType(p))** ⊑ $S_{n'}$
   $S_{n'}$ ∩ **SubTypes(StaticType(ret))** ⊑ S_m

(**p** denotes the parameter of **n'**, and **ret** denotes the return of **n'**)

XTA

4. for each method **m** ∈ **R**,
each field read **x = y.f** in **m**
   S_f ⊑ S_m

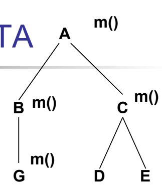
5. for each method **m** ∈ **R**,
each field write **x.f = y** in **m**
   S_m ∩ **SubTypes(StaticType(f))** ⊑ S_f

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




```
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”

```
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);
   }
}
```

Boolean Expression Hierarchy: RTA vs. XTA vs. "Ground Truth"

```
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"

```
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);
}
```