Points-to Analysis

Welf Löwe and Jonas Lundberg
Welf.Lowe@lnu.se
Jonas.Lundberg@lnu.se

Outline

- Call graph construction (simple PTA)
- Classic P2A
- P2A using simulated execution

Call / Member Reference Graph

- A Call Graph is a rooted directed graph where the nodes represent methods and constructors, and the edges represent possible interactions (calls):
  - from a method/constructor (caller) to a method/constructor (callee).
  - root of the graph is the main method.
- Generalization: Member Reference Graph also including fields (nodes) and read and write accesses (edges).

Proper Call Graphs

- A proper call graph is in addition
  - Conservative: Every call \( A.m() \rightarrow B.n() \) that may occur in a run of the program is a part of the call graph
  - Connected: Every member that is a part of the graph is reachable from the main method
- Notice
  - We may have several entry points in cases where the program in question is not complete. E.g., an implementation of the \texttt{ActionListener} interface will have the method \texttt{actionPerformed} as an additional entry point if we neglecting the \texttt{javax.swing} classes.
  - In general, it is hard to compute, which classes/methods may belong to a program because of dynamic class loading.

In reality ...

- The actual implementation of a call graph algorithm involves a lot of language specific considerations and exceptions to the basic rules. For example:
  - Field initialization and initialization blocks
  - Exceptions
  - Calls involving inner classes often need some special attention.
  - How to handle possible call back situations involving external classes.

Why computing Call Graphs

- Elimination of dead code
  - classes never loaded, no objects created from, and
  - methods never called, and
  - branches never used in any program run
- Elimination of polymorphism (usage refers to a statically known class, not to its super-class nor siblings in the class hierarchy)
- Detection of aliases (usages refer to the same object) and strangers (usages are guaranteed not to refer to the same object)
- Detection of singletons (usage refers to a single object, not to a set of objects)
- ...
Call Graphs: The Basic Problem

- The difficult task of any call graph construction algorithm is to approximate the set of members that can be targeted at different call sites.
- What is the target of call site \texttt{a.m()}?
- Depends on classes of objects potentially bound to designator expression \texttt{a}?
- Not decidable, in general, because:
  - In general, we do not have exact control flow information.
  - In general, we cannot resolve the polymorphic calls.
  - Dynamic class loading. This problem is in some sense more problematic since it is hard to make useful conservative approximations.

Declared Target

- We say that the declared target of \texttt{a.m()} occurring in a method \texttt{X.x()} is the method \texttt{m()} in the declared type of the variable \texttt{a} in the scope of \texttt{X.x()}.
- When using declared targets, connectivity can be achieved by …
  - … inserting (virtual) calls from super to subtype method declarations
  - … keeping (potentially) dynamically loaded method nodes reachable from the main method (or as additional entry points).

Discussion

- Class objects (static objects) are treated as objects
- Stack objects are considered part of \texttt{this}
  - Let \texttt{a} be a local variable or parameter, resp.
  - \texttt{a.m()} and \texttt{this.a.m()}, resp., is a usage of whatever \texttt{a} contains (target), i.e. \texttt{N(a)}, in whatever \texttt{this} contains (source), i.e. \texttt{N(this)}.

Generalized Call Graphs

- A call graph is a directed graph \( G=(V, E) \)
  - vertices \( V \) are pairs of classes \texttt{Class} and methods / constructors / fields \( m \)
  - edges \( E \) represent usage: let \( a \) and \( b \) be two objects; \( a \) uses \( b \) (in a method / constructor execution) \( a \) occurs a call / access to a method / constructor / field \( y \) of \( b \) \( \Rightarrow (\texttt{Class}(a), x, \texttt{Class}(b), y) \in E \)
- An generalized call graph is a directed graph \( G=(V, E) \)
  - vertices \( V \) are pairs of finite abstractions of runtime objects \( o \) using a so called called name schema \( N(o) \) and methods / constructors / fields \( m \)
  - edges \( E \) represent usage: let \( a \) and \( b \) be two objects; \( a \) uses \( b \) (in a method / constructor execution) \( a \) occurs a call / access to a method / constructor / field \( y \) of \( b \) \( \Rightarrow (N(a), x, N(b), y) \in E \)
- A name schema \( N(a) \) is an abstraction function with finite co-domain
  - \texttt{Class(o)} is a special name schema and, hence, describes a special call graph

Name Schemata

- One can abstract from objects by distinguishing:
  - Just heap and stack (decidable, not relevant)
  - Objects with same class (not decidable, relevant, efficient approximations)
  - Objects with same class but syntactic different creation program point (not decidable, relevant, expensive approximations)
  - Objects with same creation program point but with syntactic different path to that creation program point (not decidable, relevant, approximations exponential in execution context)
  - Different objects (not decidable)
  - …

Decidability

- Not decidable in general: reduction from termination problem
  - Add a new call (not used anywhere else before the program exit)
  - If I could compute the exact call graph, I knew if the program terminates or not
- Decidable if name schema abstract enough (then not relevant in practice)
Approximations

- Simple conservative approximation
  - from static semantic analysis
  - declared class references in a class \( A \) and their subtypes are potentially uses in \( A \)
  - \( a.x \) really uses \( b.y \) \( \Rightarrow (N(a)x, N(b)y) \in E \)
- Simple optimistic approximation
  - from profiling
  - actually used class references in an execution of class \( A \) (a number of executions) are guaranteed uses in \( A \)
  - \( a.x \) really uses \( b.y \) \( \Leftarrow (N(a)x, N(b)y) \in E \)

Simplification

For a first try, we consider only one name schema:
- Distinguish objects of different classes / types
- Consequently, a call graph is ...
  - a directed graph \( G=(V, E) \)
  - vertices \( V \) are pairs of classes and methods / constructors / fields
  - edges \( E \) represent usage: let \( A \) and \( B \) be two classes: \( A.x \) uses \( B.y \) (i.e., an instance of \( A \) executes \( x \) using an instance of \( B \)) \( \Rightarrow (A.x, B.y) \in E \)
- Not decidable still, we discuss optimistic and conservative approximations

Algorithms to discuss

All algorithms these are conservative:
- Reachability Analysis – RA
- Class Hierarchy Analysis – CHA
- Rapid Type Analysis – RTA
- …
- (context-insensitive) Control Flow Analysis – 0-CFA
- (\( k \)-context-sensitive) Control Flow Analysis – \( k \)-CFA

Reachability Analysis – RA

- Worklist algorithm maintaining reachable methods
  - initially \( \text{main} \) routine in the \( \text{Main} \) class is reachable
- For this and the following algorithms, we understand that
  - Member (field, method, constructor) names \( n \) stand for complete signatures
  - \( R \) denotes the worklist and finally reachable members
  - \( R \) may contain fields and methods/constructors. However, only methods/constructors may contain other field accesses/call sites for further processing.
- RA:
  - \( \text{Main.main} \in R \) (maybe some other entry points too)
  - \( M.m \in R \) and \( e.n \) is a field access / call site in \( m \) \( \Rightarrow \forall N \in \text{Program}: N.n \in R \land (M.m, N.n) \in E \)

Example

```java
public class Delegation {
    public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();
    }
    public static void n() {
        new C().m();
    }
}

interface I {
    public String strI = "Printing I string";
    public void m();
}

class A implements I {
    public void m() {System.out.println(strI);}
}

class B extends A {
    public B() {super();}
    public void m();
}

class C extends A {
    public void m() {System.out.println("Printing C string");}
}
```

Example
RA on Example

Delegation.main → B.new → A.init → Limit

Delegation.n → C.new

Example

```java
public class Delegation {
    public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();
    }
    public static void n() {
        new C().m();
    }
}
```

Class Hierarchy Analysis – CHA

- Refinement of RA
- `Main.main ∈ R`
- `M.m ∈ R`
  - `e.n` is a field access / call site in `M.m`
  - `type(e)` is the static (declared) type of access path expression `e`
  - `subtype(type(e))` is the set of (declared) sub-types of `type(e)`

```
⇒ ∀ N ∈ subtype(type(e)): N.n ∈ R ∧ (M.m, N.n) ∈ E
```

CHA on Example

Delegation.main → B.new → A.init → Limit

Delegation.n → C.new

Rapid Type Analysis – RTA

- Still simple and fast refinement of CHA
- Maintains reachable methods `R` and instantiated classes `S`
- Fixed point iteration: whenever `S` changes, we revisit the worklist `R`

```
⇒ ∀ N ∈ subtype(type(e)) ∈ S ∧ N.n ∈ R ∧ (M.m, N.n) ∈ E
```

```
⇒ ∀ N ∈ subtype(type(e)) : N.n ∈ R ∧ (M.m, N.n) ∈ E
```
Example

```java
public class Delegation {
  public static void main(String args[]) {
    A i = new B();
    i.m();
    Delegation.n();
  }
  public static void n() {
    new C().m();
  }
}
```

RTA on Example

Context-Insensitive Control Flow Analysis – 0-CFA

- RTA assumes that any constructed class object of a type can be bound to an access path expression of the same type
- Considering the control flow of the program, the set of reaching objects further reduces
- Example:

```
main() { class A {
  A a = new A();
  public void m() {...
  a.m();
}
sub();
}
sub(){ class B extends A {
  A a = new B();
  public void m() {...
  a.m();
}
}
```

Context-Sensitive Control Flow Analysis – k-CFA

- 0-CFA merges objects that can reach an access path expression (designator) via different call paths
- One can do better when distinguishing the objects that can reach an access path expression via paths differing in the last $k$ nodes of the call paths
- Example:

```
main() { class A {
  A a = new A();
  3. dispatch(a);
}
sub();
}
sub(){ class B extends A {
  A a = new B();
  3. dispatch(a);
}
}
```

Control Flow Analysis

- Requires data flow analysis
- 0-CFA: has already high memory consumption in practice (still practical)
- $k$-CFA: is exponential in $k$
  - Requires a refined name schema (and, hence, even more memory)
  - Does not scale in practice (if extensively used)
  - Solutions discussed later today
- One idea (current research):
  - Make $k$ adaptive over the analysis
  - Focus on specific program parts
  - Reduce $k$ to max 1
Order on Algorithms

- Increasing complexity
- Increasing accuracy

RA \longrightarrow CHA \longrightarrow RTA \longrightarrow 0-CFA \longrightarrow 1-CFA \longrightarrow \ldots

Complexity & Accuracy

- Analyses between RTA and 0-CFA?

Analyses Between RTA and 0-CFA

- RTA uses one set \( S \) of instantiated classes
- Idea:
  - Distinguish different sets of instantiated classes reaching a specific field or method
  - Attach them to these fields, methods
  - Gives a more precise "local" view on object types possibly bound to the fields or methods
- Regards the control flow between methods but
- Disregards the control flow within methods
- Fixed point iteration

Notations

- Subtypes of a set of types:
  \[
  \text{subtype} \left( S \right) := \bigcup_{N \in S} \text{subtype} \left( N \right)
  \]
- Set of parameter types \( \text{param} \left( m \right) \) of a method \( m \): all static (declared) argument types of \( m \) excluding type (this)
- Return type \( \text{return} \left( m \right) \) of a method \( m \): the static (declared) return type of \( m \)

Separated Type Analysis – XTA

- Separate type sets \( S_m \) reaching methods \( m \) and fields \( x \) (treat fields \( x \) like methods pairs \( \text{set}_x, \text{get}_x \))
- \( \text{Main.main} \in R \)
- \( \text{M.m} \in R \)
- For all class (static) methods \( s \): \( \text{class} \left( s \right) \in S_m \)
  - new \( N \) is a constructor call site in \( M.m \)
    \( \Rightarrow N \in S_w \land N.new \in R \land (M.m, N.new) \in E \)
  - \( e.n \) is a field access / call site in \( M.m \)
    \( \Rightarrow \forall N \in \text{subtype} \left( \text{type} \left( e \right) \right) \land N \in S_m \land (M.m, N.n) \in E \)
    \( \Rightarrow \text{result} \left( N.n \right) \in S_m \land \text{result} \left( N.n \right) \subseteq S \land (M.m, N.n) \in E \)

Example

```java
public class Delegation {
    public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();
    }
    public static void n() {
        new C().m();
    }
}

interface I {
    public String strI = "Printing I string";
    public void m();
}

class A implements I {
    public void m() {System.out.println(strI);}
}

class B extends A {
    public B() {super();}
}

class C extends A {
    public void m() {System.out.println("Printing C string");}
}
```

XTA on Example
**Source**


- David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers: *Call Graph Construction in Object-Oriented Languages*. OOPSLA 1997.

**Conclusion on Call Graphs so far**

- Approximations
  - Relatively fast, feasible for large systems
  - Relatively imprecise, conservative
- What is a good enough approximation of certain client analyses
- Answer depends on client analyses (e.g., different answers for software metrics and clustering vs. program optimizations)
Outline

- Call graph construction (simple PTA, name schema: classes 4 objects)
- Classic P2A (name schema: creation sites 4 objects)
- P2A using simulated execution

Client-Applications of Points-to Analysis

- Points-to results can be used as input for a number of compiler related activities. We refer to these activities as client-applications.
  - Resolve call sites and field accesses: Given the points-to set \( Pt(a) \) it is easy to resolve possible targets of a call site \( a.m() \) and field accesses \( a.f \).
  - A call site \( a.m() \) is said to be statically decidable if only one target is possible (i.e. \( |Pt(a)| = 1 \)). This information can be used to replace virtual calls (requires dynamic lookup) with direct calls (no lookup necessary).
  - Inter-procedural control-flow: Similarly, resolving call sites and field accesses is a prerequisite for any analysis that requires inter-procedural control-flow information. For example, constant folding and common sub-expression elimination.
  - Synchronization Removal: In multi-threaded programs each object has a lock to ensure mutual exclusion. If we can identify thread-local objects (objects only accessed from within the thread) their locks can be removed and execution time reduced.
  - Static Garbage Collection: Method-local objects (objects only referenced from within a given method) can be put on the stack rather than the heap and these objects will be automatically de-allocated once a method execution been completed.

Classic P2A: Introduction

- We try to find all objects that each reference variable may point to (hold a reference to) during an execution of the program.
- Hence, to each reference variable \( v \) in a program we associate a set of objects, denoted \( Pt(v) \), that contains all the objects that variable \( v \) may point to. The set \( Pt(v) \) is called the points-to set of variable \( v \).
- Example:
  ```java
  A a,b,c;
  X x,y;
  s1: a = new A(); // Pt ( a ) = {o1}
  s2: b = new A();
      b = a;
      // Pt ( b ) = {o1 , o2}
  c = b;
      // Pt ( c ) = {o1 , o2}
  ```
  Here \( o1 \) means the object created at allocation site \( s1 \).
- After a completed analysis, each variable \( v \) is associated with a points-to set \( Pt(v) \) containing a set of objects that it may refer to.

Outline of the classic approach

Points-to analysis (any data-flow analysis) requires:
1. Deciding upon a set of data values (analysis value domain \( U \))
2. Constructing a data-flow graph which indicates the flow of data.
3. Initialize the graph with data.
4. Propagate the data along the edges in the data-flow graph until a fixed point is reached.

Name Schema revisited

- The number of objects appearing in a program is in general infinite (countable), hence, we don’t have a well-defined set of data values.
- For example, consider the following situation
  ```java
  while ( x > y ) {
    A a = new A();
    …
  }
  ```
  The number of \( A \) objects is in cases like this impossible to decide. (Think if \( x \) or \( y \) depends on some input values).
- From now on, each object creation point (new \( A() \), \( a.clone() \), “hello”) represents a unique object (identified by the source code location).
- Again, many run-time objects are mapped to a single abstract object.

Object Transport as Set Constraints

- Objects can flow between variables due to assignments and calls. Calls will be treated shortly.
- Certain statements generates constraints between points-to sets. We will consider:
  ```java
  l = r \Rightarrow Pt(r) \subseteq Pt(l) \quad \text{(Assignment)}
  l = \text{new } A(); \Rightarrow \{o1\} \subseteq Pt(l) \quad \text{(Allocation)}
  ```
  That is, each assignment can be interpreted as a constraint between the involved points-to sets.
- Each statement in the program will generate constraints, as before in DFA, we will have a system of constraints.
- We are looking for the minimum solution (minimum size of the points-to sets) that satisfies the resulting system of constraints, i.e., the minimum fixed point of the dataflow equations.
Example

A Simple Program

```java
public & methodX(A param) {
    A a1 = param;
    s1 : A a2 = new A();
    A a3 = a1;
    a3 = a2;
    return a3;
}
```

Generated set constraints

```java
1: Pt(param) ⊆ Pt(a1)
2: o1 ∈ Pt(a2)
3: Pt(a1) ⊆ Pt(a3)
4: Pt(a2) ⊆ Pt(a3)
```

Object Transport in terms of P2G edges

- Each constraint can be represented as a relation between nodes in a graph.
- A Points-to Graph P2G is a directed graph having variables and objects as nodes and assignments and allocations as edges
- P2G initialization: \( o1 \rightarrow a \), add \( o1 \) to \( Pt(i) \).
- P2G propagation: \( F \rightarrow a \), add let \( Pt(i) = Pt(i) \cup Pt(r) \)

Analysis Approximation: Flow-Insensitivity

- An analysis is flow-sensitive if the order of execution of statements is taken in account.
- In the code example below illustrates flow-sensitivity.

```
(1) s1 : f = new A()
(2) a = f
(3) s2 : f = new A()
        // insensitive: Pt(f)={o1,o2}
        // sensitive: Pt(f)={o1}
(4) b = f
        // insensitive: Pt(b)={o1,o2}
        // sensitive: Pt(b)={o2}
```

Our approach would have generated the following set of constraints

\( o1 \in Pt(f) \), \( o2 \in Pt(f) \), \( o2 \in Pt(f) \), \( Pt(f) \subseteq Pt(b) \)

Constraints (1) and (3) yield \( Pt(f)={o1,o2} \) and consequently that both \( a \) and \( b \) have \( Pt(o1,o2) \).

Thus, a consequence of using a set constraint approach is flow-insensitivity.

A flow-sensitive analysis requires that each definition of a variable has a node and a points-to set. This makes the graph much larger and the analysis more costly.

Forward reference Static Single Assignment representation

Method Calls: Object Transfer

```
s1 : A a = new A() // o2 → a
s2 : X x1 = new X() // o2 → x2
        a.loadX(x1) // o2 → x4, x4 → x3, x3 → f
        X x2 = a.loadX() // f → x2
classX {
    X f;
    private void setX(X x3) {if (x3);}
    private X getX() {return this.getX();}
    public X loadX() {return this.getX();}
}
```

Involved object transport

- Argument passing, i.e., assigning arguments to parameters (e.g. \( x1 \rightarrow x4 \)).
- A call \( a.m() \) involves an implicit assignment \( a=m(). \)
- The return assignment involves two implicit steps too (\( f \rightarrow x2 \)).
- The above approach contains many implicit steps that are hard to perform automatically.
- The procedural method representation gives a more explicit view of the object transport.

Representation of Methods

**OO Definition**

```java
classA {
    public R mName(P1 p1, P2 p2) {
        return Rexpr;
    }
}
```

**Procedural Definition**

```java
public R mName(P1 p1, P2 p2, R ret) {
    return Rexpr;
}
```

**OO Invocation**

```java
l = a.mName(x,y);  
```

**Procedural Invocation**

```java
mName(a,x,y,l);  
```

Advantage of Procedural Representation

- Given a call site \( l=r0.m(r1,\ldots,rn) \)
  - Represented as \( m(r0,r1,\ldots,rn,0) \)
  - Targeted at method public R mName(P1 p1, P2 p2) in classA
  - Represented as \( n(A this,P1 p1,\ldots,Pn pn,R ret) \)
- We add the following P2G edges
  - \( r0 \rightarrow this \), \( r1 \rightarrow p1 \), \( \ldots \), \( rn \rightarrow pn \), \( ret \rightarrow l \)
- Thus, each resolved call site results in a well-defined set of inter-procedural P2G edges.
We refer to P2Gs where no more edges are to be added as complete.

**Advantage:** The precision of the externally derived call graph influences the points-to-analysis.

We refer to P2Gs where no more edges are to be added as complete. Complete P2Gs are much easier to handle as will be discussed shortly.

The trick is to narrow down the number of possible call targets.

**Resolving Call Targets**

- The procedural method representation makes it quite easy to generate a set of P2G edges once the target method has been identified. The problem is to find target methods.
- Static calls and constructor calls are easy; they always have a well-defined target method.
- Virtual calls are much harder; to accurately decide the target of a call site during program analysis is in general impossible.
- Any points-to analysis involves some kind of conservative approximation where we take into account all possible targets.

**Static Dispatch**

Given a conservative call graph, we can construct a function `staticDispatch(A this, X x)` that provides us with a set of possible target methods for any given call site `a.m()`.

We can then proceed as follows:

- **Advantage:** We can immediately resolve all call sites and add corresponding P2G edges.
- **Disadvantage:** The precision of the externally derived call graph influences the points-to-analysis.

Given a conservative call graph we can construct a function `staticDispatch(A this, X x)` that provides us with a set of possible target methods for any given call site `a.m()`.

We can then proceed as follows:

- **Static Dispatch:** Given an externally derived conservative call graph (discussed before) we can approximate the actual targets of any call site in a program. By using such a call graph we can associate each call site `a.m()` with a set of pre-computed target methods `T_1.m(),...T_n.m()`.
- **Dynamic Dispatch:** By using the currently available points-to-set `Pt(a)` itself, we can, for each object in the set, find the corresponding dynamic class and, hence, the target method definition of any call site `a.m()`.

Two approaches to resolve a call site `a.m()`

- **Static Dispatch:** Given an externally derived conservative call graph (discussed before) we can approximate the actual targets of any call site in a program. By using such a call graph we can associate each call site `a.m()` with a set of pre-computed target methods `T_1.m(),...T_n.m()`.
- **Dynamic Dispatch:** By using the currently available points-to-set `Pt(a)` itself, we can, for each object in the set, find the corresponding dynamic class and, hence, the target method definition of any call site `a.m()`.

- **Dynamic Dispatch**

Given the points-to set `Pt(a)` of a variable `a` we can resolve the targets of a call site `a.m()` using a function `dynamicDispatch(A, m)` that returns the method executed when we invoke the call `a.m()` with signature `m` on an object `o` of type `A`.

We can then proceed as follows:

- **Advantage:** We avoid using an externally defined call graph.
- **Disadvantage:** The P2G is not complete since we initially don’t know all members of `Pt(a)`.
- Hence, the P2G will change (additional edges will be added) during analysis.
Propagating a Complete P2G

- In this approach we use work list to store variable nodes that need to be propagated.
  1. For each variable \( v \) let \( \mathcal{P}(v) = \emptyset \) // \( O(\#v) \)
  2. For each allocation edge \( o \rightarrow v \) do // \( O(\#o) \)
     (a) let \( \mathcal{P}(v) = \mathcal{P}(v) \cup \{ o \} \)
     (b) add \( v \) to worklist
  3. Repeat until worklist empty // \( O(\#v \times \#o) \)
     (a) Remove first node \( v \) from worklist
     (b) For each edge \( p \rightarrow q \) do // \( O(\#v) \)
        i. Let \( \mathcal{P}(q) = \mathcal{P}(q) \cup \mathcal{P}(p) \)
        ii. If \( \mathcal{P}(q) \) has changed, add \( q \) to worklist

- Time complexity: Let \( k \) be the number of variable nodes and \( s \) the number of (abstract) objects.
- A node is added to the work list each time it is changed.
- In the worst case this can happen \( k \times s \) times for each node, thus, we have \( O(k \times s \times \#o) \) number of work list iterations.
- Each such iterations may update every other variable node (hence \( k \times s \times \#o \) within the loop). Thus, an upper limit is \( O(k \times s \times \#o) \).

Optimizing the Analysis

- The high time complexity \( O(k \times s \times \#o) \) encourages optimizations. Optimizations can basically be done in two different ways:
  1. Removal of strongly connected components
  2. Removal of single dominated subgraphs.
- We can speed up the propagation algorithm by processing the nodes in a more clever ordering:
  1. Topological node ordering.
  2. Other optimizations are possible all three are simple and effective.

Previous Example Revisited: Results of Points-to Analysis

```java
class Main {
    static procedure main(Main this, String[] args) {
        s1 = A1 = new A1(); // Pt(A1) = {o1}
        s2 = x1 = new X1(); // Pt(x1) = {o2}
        store(x1, A1); // Pt(A1) = {o2}
        load(x1, A2); // Pt(A2) = {o3}
        s3 = A2 = new A2(); // Pt(A2) = {o3}
        s4 = x2 = new X2(); // Pt(x2) = {o4}
        load(x2, A1); // Pt(A1) = {o4}
        // Pt(x2) = {o4}
    }
}

class A1 {
    procedure loadX(A this4, X r2) {getX(this4, r2);}
    procedure storeX(A this3, X x4) {setX(this3, x4);}
    procedure getX(A this2, X r1) {r1 = f;}
    procedure setX(A this1, X x3) {f = x3;}
    X f;
}

class A2 {
    procedure loadX(A this4, X r2) {getX(this4, r2);}
    procedure storeX(A this3, X x4) {setX(this3, x4);}
    procedure getX(A this2, X r1) {r1 = f;}
    procedure setX(A this1, X x3) {f = x3;}
    X f;
}

class X1 {
    // Pt(this1) = {o1, o3}
    procedure loadX(A this4, X x3) {x3 = t3; // Pt(x3) = {o2, o4}
        procedure newX1 this3, X x4 {getX(this3, x4); // Pt(x4) = {o2, o4}
        procedure newX2 this3, X x4 {getX(this3, x4); // Pt(x4) = {o2, o4}
    }
}

class X2 {
    // Pt(this2) = {o1, o3}
    procedure loadX(A this4, X x3) {x3 = t3; // Pt(x3) = {o2, o4}
        procedure newX1 this3, X x4 {getX(this3, x4); // Pt(x4) = {o2, o4}
        procedure newX2 this3, X x4 {getX(this3, x4); // Pt(x4) = {o2, o4}
    }
}
```

Limitations of Classic Points-to Analysis

- In the previous example we found that \( Pt(A, 2) = \{o_2, o_4\} \). However, from the program code it is obvious that we have two instances of class \( A(A_1) \) and \( A(A_2) \) and that \( Pt(A(A_1), 1) = \{o_2\} \) whereas \( Pt(A(A_2), 1) = \{o_4\} \). Hence by having a common points-to set for field variables in different objects the different object states are merged.
- Consider two \( List \) objects created at different locations in the program. We use the first list to store \( String \) objects and the other to store \( Integer \) objects. Using ordinary points-to analysis we would find that both these list store both strings and objects.
- Conclusion: Classic points-to-analysis merges the states in objects created at different locations and, as a result, can't distinguish their individual states and content.
- Context-sensitive approaches would let each object has its own set of fields. This would however correspond to object/method inlining and increase the number of P2G nodes exponentially and reduce the analysis speed accordingly.
- Flow-sensitive would increase precision as well, at the price of adding new nodes for every definition of a variable. Once again, increased precision at the price of performance loss.
- The trade-off between precision and performance is a part of everyday life in data-flow analysis. In theory we know how to increase the precision, unfortunately not without a significant performance loss.

Outline

- Call graph construction (simplistic P2A)
- Classic P2A
- P2A using simulated execution