Inter-Procedural Analysis and Points-to Analysis

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

Outline

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise, not today – requires SSA)

Inter-Procedural Analysis

- What is inter-procedural dataflow analysis
  - DFA that propagates dataflow values over procedure boundaries
  - Finds the impact of calls to caller and callee
- Tasks:
  - Determine a conservative approximation of the called procedures for all call sites
    - Referred to as Call Graph construction (more general: Points-to analysis)
  - Tricky in the presence of function pointers, polymorphism and procedure variables
  - Perform conservative dataflow analysis over basic-blocks of procedures involved
- Reason:
  - Allows new analysis questions (code inlining, removal of virtual calls)
  - For analysis questions with intra-procedural dataflow analyses, it is more precise (dead code, code parallelization)
- Precondition:
  - Complete program
  - No separate compilation
  - Hard for languages with dynamic code loading

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).

Techniques for Inter-Procedural Analysis

- Intra-procedural analysis on an inlined basic block graphs (textbook approach)
- Simulated execution

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 ActionListener interface will have the method actionPerformed as an additional entry point if we neglecting the java.swing classes.
    - Libraries miss a main method
  - In general, it is hard to compute, which classes/methods may belong to a program because of dynamic class loading.
“Inlined” basic block graphs

- Given call graph and a bunch of procedures each with a basic block graph
- Inline basic block graphs
  - Split call nodes (and hence basic blocks) into callBegin and callEnd nodes
  - Connect callBegin with entry blocks of procedures called
  - Connect callEnd with exit blocks of procedures called
  - Entry (exit) block of main method gets start node of forward (backwards) dataflow analysis
- Polymorphism is resolved by explicit dispatcher or by several targets
- Inter-procedural dataflow analysis now possible as before ab intra-procedural analysis

Example Program

```java
public class One {
    public static void main(String[] args) {
        int x=0; x=r(x); x=q(x); x=r(x);
        System.out.println("Result: " + x);
    }

    static int r(int x) {
        if (x==1) x=s(x); return(x);
    }

    static int q(int x) {
        if (x==1) x=s(x); else x=t(x); return(x);
    }

    static int s(int x) {
        if (x==0) x=r(x); return(x);
    }

    static int t(int x) {
        return(x+1);
    }
}
```

Example: Unrealizable Path

- Data gets propagated along path that never occur in any program run:
  - Calls to one method returning to another method
  - CallBegin → Method Start → Method End → CallEnd
  - Makes analysis conservative
  - Still correct
  - (And still more precise than corresponding intra-procedural analyses)
Simulated Execution

- Starts with analyzing main
- Interleaving of analyze method and the transfer function of calls’
- A method (intra-procedural analysis):
  - propagates data values along the edges in basic-block graph
  - updates the analysis values in the nodes according to their transfer functions
  - if node type is a call then ...
- Calls’ transfer function and only if the target method input changed:
  - Interrupts the processing of a caller method
  - Propagates arguments \( v_1 \ldots v_n \) to the all callees
  - Processes the callees (one by one) completely
  - Iterate to local fixed point in case of recursive calls
  - Propagates back and merges (supremum) the results \( r \) of the callees
  - Continue processing the caller method ...

Comparison

- Advantages of Simulated Execution
  - Fewer non realizable path, therefore:
    - More precise
    - Faster
- Disadvantages of Simulated Execution
  - Harder to implement
  - More complex handling of recursive calls
  - Leaves theory of monotone DFM and Abstract Interpretation

Outline

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise, not today – requires SSA)

Why are we interested?

- Elimination of dead code i.e., classes never loaded, no objects created from, and methods never called.
- 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 access does not need to be locked.
- Static Garbage Collection: Method-local objects (objects only referenced from within a given method) can be allocated on the stack rather than the heap and these objects will be automatically de-allocated once a method execution has been completed.
- Elimination of polymorphism: usage refers to a statically known method i.e., only one target is possible.

Call Graph Construction 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.

... and more

- Resolving call sites and field accesses i.e. constructing a precise call graph is a prerequisite for any analysis that requires inter-procedural control-flow information. For example, constant folding and common sub-expression elimination.
- Detection of aliases (usages refer to the same object) which is a prerequisite for protocol checking.
- Detection of design patterns (e.g., singletons usage refers to a single object, not a set of objects of the same type) and anti-patterns.
  - ...
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()} 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 $V \subseteq \text{Class} \times \text{Method/Constructor/Field}$
  - Edges \texttt{E} represent usage: let \texttt{a} and \texttt{b} be two objects: \texttt{a} uses \texttt{b} (in a method / constructor execution \texttt{x} of \texttt{a} occurs a call / access to a method / constructor / field \texttt{y} of \texttt{b}) $\iff (\text{Class}(a),x,\text{Class}(b),y) \in \text{E}$
- An generalized call graph is a directed graph $G^\prime=(V,E)$
  - Vertices $V \subseteq \text{N}(o) \times \text{Method/Constructor/Field}$ are pairs of finite abstractions of runtime objects \texttt{o} using a so called called name schema \texttt{N(o)} and methods / constructors / fields \texttt{E}.
  - Edges \texttt{E} represent usage: let \texttt{a} and \texttt{b} be two objects: \texttt{a} uses \texttt{b} (in a method / constructor execution \texttt{x} of \texttt{a} occurs a call / access to a method / constructor / field \texttt{y} of \texttt{b}) $\iff (\text{N(a)},x,\text{N(b)},y) \in \text{E}$
- A name schema \texttt{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 new 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
  - Formally, $N(o) = \text{Class(o)}$
  - 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 method / constructor / field $y$ instance of $B$) \( \Rightarrow (A.x, B.y) \in E \)

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
    - $M.m$ 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

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

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

Delegation.main B.new A.init I.init
  I.m
A.m
B.m
  PrintStream.println
  C.m
C.new
Delegation.n

Example

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");}
}

CHA on Example

Delegation.main B.new A.init I.init
  I.m
A.m
B.m
  PrintStream.println
  C.m
C.new
Delegation.n

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 \( R \)

Main.main \( \in R \)

For all class (static) methods \( s : \text{class}(s) \in S \)

- new \( N \) is a constructor call site in \( M.m \)
  \[ \Rightarrow N \in S \times \text{new} \in R \times (M.m, N\text{new}) \in E \]

- \( e \) is a field access / call site in \( M.m \)
  \[ \Rightarrow \forall N \in \text{subtype}(\text{type}(e)) : N \in S \times (M.m, N.e) \in E \]
Example

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

```java
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");
    }
}
```

RTA on Example

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

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:

```
public class X {
    public static void dispatch(A a) { a.m(); }
}

main() {
    class A {
        public void n(){...}
    }

    class B extends A {
        public void n(){...}
    }

    class C extends A {
        public void n(){...}
    }

    X.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

RTA — CHA — RTA — 0-CFA — 1-CFA — ...

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

Separated Type Analysis – XTA

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

Example

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

```
public class Delegation {
    public static void main(String args[]) {
        A a = new B();
        a.m();
    }
    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");}
}
Increasing complexity

- Number of type separating sets \( S \) (\( M \) number of methods, \( F \) number of fields):
  - CHA: 0
  - RTA: 1
  - XTA: \( M + F \)
- Practical observations on benchmarks:
  - All algorithms RA...XTA scale (1 Mio. Loc)
  - XTA one order of magnitude slower than RTA
  - Correlation to program size rather weak

Increasing precision

- Practical observations on benchmarks:
  - RTA as baseline: all instantiated (wherever) classes are available in all methods
  - XTA on average:
    - Only ca. 10% of all classes are available in methods
    - < 3% fewer reachable methods
    - > 10% fewer call edges
    - > 10% more monomorphic call targets

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

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise, not today – requires SSA)

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
  class A {
  }
  while ( x > y ) {
      x = x - y;
  }
  ```
  The number of objects appearing in a program is in general infinite (countable). However, in many cases, we can associate a fixed size (finite) set of objects with a variable. (Think if \( x \) or \( y \) depends on some input values).
- From now on, each object is associated with a single abstract object.
- Finitely many abstract objects are mapped to a single abstract object.
- Each statement in the program will generate constraints, as before equation in DFA, we will have a system of constraints.
- For example, consider the following situation:
  ```java
  class A {
  }
  while ( x > y ) {
      x = x - y;
  }
  ```
  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
  class A {
  }
  while ( x > y ) {
      x = x - y;
  }
  ```
  The number of objects appearing in a program is in general infinite (countable).

Example

A Simple Program
```java
public A methodX(A param) {
    A a1 = param;
    A a2 = new A();
    A a3 = a1;
    a3 = a2;
    return a3;
}
```
Generated set constraints
```
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 is our data-flow graph and the objects are our data values to be propagated.
- P2G initialization: \( \forall o_i \rightarrow 1, \text{ add } o_i \text{ to } Pt(1) \).
- P2G propagation: \( \forall c \rightarrow 1, \text{ add let } Pt(1) = Pt(1) \cup Pt(x) \).

Object Transport as Set Constraints

- Objects can flow between variables due to assignments and calls. Calls will be treated shortly.
- Certain statements generate constraints between points-to sets. For example, consider the following situation:
  ```java
  class A {
  }
  while ( x > y ) {
      x = x - y;
  }
  ```
  Here \( o_i \) means the object created at allocation site \( a_i \).
- 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.

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(); // Pt(b) = {o2}
  c = b; // Pt(c) = {o1, o2}
  ```
- Here \( o_i \) means the object created at allocation site \( a_i \).
- 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.

Generated set constraints
```
1: Pt(param) ⊆ Pt(a1)
2: o1 ⊆ Pt(a2)
3: Pt(a1) ⊆ Pt(a3)
4: Pt(a2) ⊆ Pt(a3)
```

Object Transport as Set Constraints

- Objects can flow between variables due to assignments and calls. Calls will be treated shortly.
- Certain statements generate constraints between points-to sets. We will consider:
  ```java
  class A {
  }
  while ( x > y ) {
      x = x - y;
  }
  ```
  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 equation 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.

Outline

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise, not today – requires SSA)
Assignments and Allocations (flowinsensitive in a method)

- Assignment and Allocation
- Constraints Pt(Pi[C, Pi]) and due to initialization a = Pt(Pi)
- Graph generated \( r \rightarrow \{i\} \) resp.
- Constraints Pt(Pi[C, Pi]) and Pt(Pi[C, Pi])
- Graph generated \( r \rightarrow \{i\} \) resp.

1. \( a = f \text{ new } a() \)
2. \( a = f() \)
3. \( f = a() \)
4. \( f = \text{ new } a() \)

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

Method Calls and Definitions (flowinsensitive between methods)

- Method Calls = \{m1(r1, r2, ...) and Definitions \( m/this, p, p,... \} \) return res
- Defines Pt(Pi[C, Pi]) and Pt(Pi[C, Pi])
- Graph generated \( r \rightarrow \{i\} \) resp.

\( x = a = \text{ new } x() \)
\( x = a = x() \)
\( x = a = x() \)

Advantage of Procedural Representation

- Given a call site \( j=x.0.m(r_1, ..., r_n) \)
- \( \text{Represented as } s(r_0, r_1, ..., r_n, l) \)
- Targeted at method \( public R mName(P1 p1, P2 p2) \) \( classA \)
- \( \text{Represented as } s(A, P1 p1, ..., Pn p, R ret) \)
- We add the following P2G edges

Previous Example Revisited / Extended

```
class Main {
    static procedure main (Main this , String[] args) {  
        \( x_1, x_2, x_3 \) = new A();  // a1 = a1
        \( x_1 = x_2 = x_3 \);  // a2 = a1
        \( x_1 = x_3 = x_1 \);  // a3 = a3
        \( x_2 = x_3 = x_1 \);  // a4 = a3
        \( loadX x_1, x_2 \);  // a1 = a3, ret2 = a2
        \( loadX x_2, x_3 \);  // a2 = a3
    }
}
```

P2G Generated

```
\( x_1, x_2, x_3 \) = new A();  // a1 = a1
\( x_1 = x_2 = x_3 \);  // a2 = a1
\( x_1 = x_3 = x_1 \);  // a3 = a3
\( x_2 = x_3 = x_1 \);  // a4 = a3
\( loadX x_1, x_2 \);  // a1 = a3, ret2 = a2
\( loadX x_2, x_3 \);  // a2 = a3
```
Resolving Call Targets

- The procedural method representation makes it quite easy to generate a set of Call Graph edges once the target method has been identified. The problem is to find the 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.
- The trick is to narrow down the number of possible call targets.

1. For each variable
   1. For each call site
     In the worst case this can happen

Time complexity: Let

In this approach we use work list to store variable nodes that need to be propagated.

We refer to P2Gs where no more edges are to be added as complete.

Disadvantage: The precision of the externally derived call graph is impossible.

Advantage: We can immediately resolve all call sites and add corresponding P2G edges.

Static Dispatch

- Given a conservative call graph we can construct a function
  staticDispatch(a, n(1)) that provides us with a set of possible target methods for any given call site a.n(1).
- We can then proceed as follows:
  for each call site
 1. let targets = staticDispatch(c.d.n(1))
 2. for each method n(A this, P1 p1, ... Pn pn, ret) targets do
    add P2G edges v0 = this, cl = p1, ... cn = pn, ret = l
 3. Advantage: We can immediately resolve all call sites and add corresponding P2G edges.
 4. Disadvantage: The precision of the externally derived call graph influences the points-to-analysis.
 5. 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.

Resolving Polymorphic Calls

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 Tm(1), ... Tm(n).
- 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 m() on an object a of type a.
- We can then proceed as follows:
  for each call site
 1. let A = signatureOf(Object)
 2. for each method signatureOf(A, P1 p1, ... Pn pn, ret) =
    dynamicDispatch(a, m)
 3. Advantage: We avoid using an externally defined call graph.
 4. Disadvantage: The P2G is not complete since we initially don’t know all members of Pt(a).
 5. Hence, the P2G will change (additional edges will be added) during analysis.

Optimizing the Analysis

- The high time complexity \(\mathcal{O}(n^2 + e)\) 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:
- Topological node ordering.
- Other optimizations are possible all three are simple and effective.

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 Pt(v) = \(\emptyset\)
  2. For each allocation edge \(a_i \rightarrow v\) do
    a. let Pt(v) = Pt(v) \cup \{a_i\}
    b. add v to worklist
  3. Repeat until worklist empty
    a. Remove first node p from worklist
    b. For each edge \(p \rightarrow q\) do
      i. Let Pt(q) = Pt(q) \cup Pt(p)
      ii. If Pt(q) has changed, add q to worklist
  4. Time complexity. Let \(n\) be the number of variable nodes and \(e\) the number of (abstract) objects.
  5. A node is added to the work list each time it is changed.
  6. In the worst case this can happen \(\mathcal{O}(e)\) times for each node, thus, we have \(\mathcal{O}(n^3 + e^2)\) number of work list iterations.
  7. Each such iteration may update every other variable node (hence \(\mathcal{O}(n^3)\) within the loop). Thus, an upper limit is \(\mathcal{O}(n^3 + e^2)\).
Previous Example Revisited: Results of Points-to Analysis

class Main {
  static procedure main (Main this, String[] args) {
    s1: A a1 = new A(); // Pt(a1) = {o1}
    s2: X x1 = new X(); // Pt(x1) = {o2}
    storeX(a1, x1);
    X x2;  // Pt(x1) = {o2, o4}
    loadX(a1, x2);
    s3: A a2 = new A(); // Pt(a2) = {o3}
    s4: X x5 = new X(); // Pt(x5) = {o4}
    storeX(a2, x5);
    loadX(a2, x2);
  }
}
class A {
  X f;  // Pt(f) = {o2, o4}
  procedure setX(A this1, X x3) { f = x3; } // Pt(this1) = {o1, o3}, Pt(x3) = {o2, o4}
  procedure getX(A this2, X r1) { r1 = f; } // Pt(this2) = {o1, o3}, Pt(r1) = {o2, o4}
  procedure storeX(A this3, X x4) { setX(this3, x4); } // Pt(this3) = {o1, o3}, Pt(x4) = {o2, o4}
  procedure loadX(A this4, X r2) { getX(this4, r2); } // Pt(this4) = {o1, o3}, Pt(r2) = {o2, o4}
}

Limitations of Classic Points-to Analysis

- In the previous example we found that Pt(A.f) = {o2, o4}. However, from the program code it is obvious that we have two instances of class A (o1 and o2) and that Pt(o1.f) = {o2} whereas Pt(o2.f) = {o4}. 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. Using ordinary points-to analysis we would find that both these lists 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 have 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-sensitivity 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

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise)