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 presents 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).

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.

Techniques for Inter-Procedural Analysis

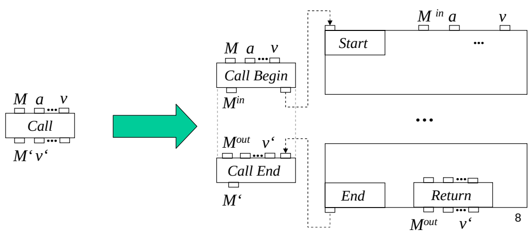
- Intra-procedural analysis on an inlined basic block graphs (textbook approach)
- Simulated execution

“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

“Inlining” of basic block graphs

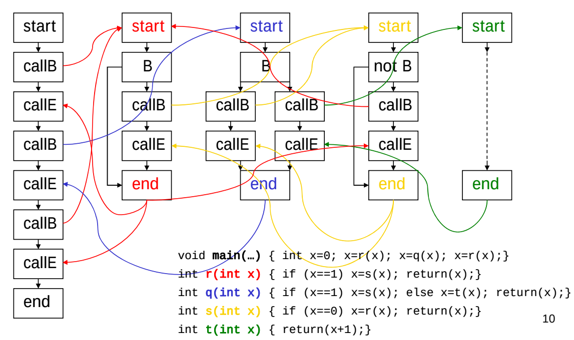
- New node: begin and end of calls distinguished
- Edges: connection between caller and callees



Example Program

```
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);
   }
}
```

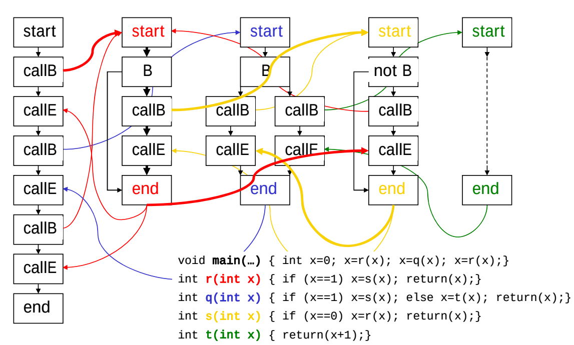

Example



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)

Example: Unrealizable Path



Simulated Execution

- Starts with analyzing main
- Interleaving of analyze method and the transfer function of calls'
- A method (intra-procedural analysis):
  - propagates data values analog 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 …

13

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

14

Outline

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

15

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.

16

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)
- …

17

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 `a.m()`
- Depends on classes of objects potentially bound to designator expression a?
- Not decidable, in general, because:
  - In general, we do not have exact control flow information.
  - In general, we can not resolve the polymorphic calls.
  - Dynamic class loading. This problem is in some sense more problematic since it is hard to make useful conservative approximations.

18

Declared Target

- We say that the declared target of `a.m()` occurring in a method `X.x()` is the method `m()` in the declared type of the variable a in the scope of `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 *this*
  - Let *a* be a local variable or parameter, resp.
  - *a.m*() is a usage of whatever *a* contains (target), i.e. $N(a)$, in whatever *this* contains (source), i.e. $N(this)$.

Generalized Call Graphs

- A call graph is a directed graph $G=(V, E)$
  - vertices $V = Class.m$ are pairs of classes *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 *x* of *a* occurs a call / access to a method / constructor / field *y* of *b*) ⇔ $(Class(a).x, Class(b).y) \in E$
- An generalized call graph is a directed graph $G=(V, E)$
  - vertices $V = N(o).m$ 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 *x* of *a* occurs a call / access to a method / constructor / field *y* of *b*) ⇔ $(N(a).x, N(b).y) \in E$
- A name schema *N* a is an abstraction function with finite co-domain
- $Class(o)$ 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, $N(o)=Class(o)$
- 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) $\Leftrightarrow (A.x, B.y) \in E$
- Not decidable still, we discuss optimistic and conservative approximations

25

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

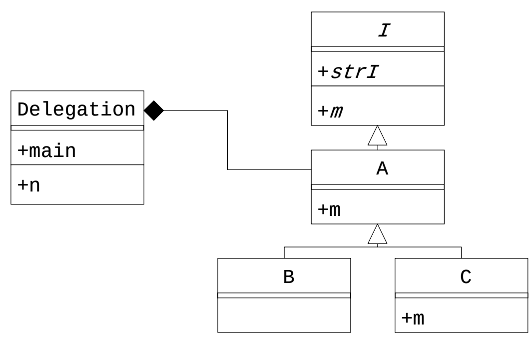
26

Reachability Analysis – RA

- Worklist algorithm maintaining reachable methods
  - initially *main* routine in the *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:
  - $Main.main \in R$ (maybe some other entry points too)
  - $M.m \in R$ and $e.n$ is a field access / call site in m $\Rightarrow \forall\, N \in Program: N.n \in R \wedge (M.m, N.n) \in E$

27

Example



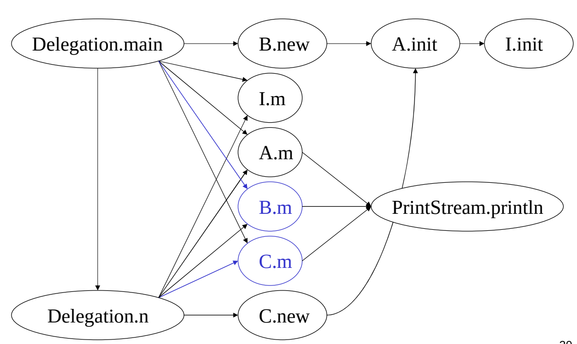
28

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

29

RA on Example



30

Class Hierarchy Analysis – CHA

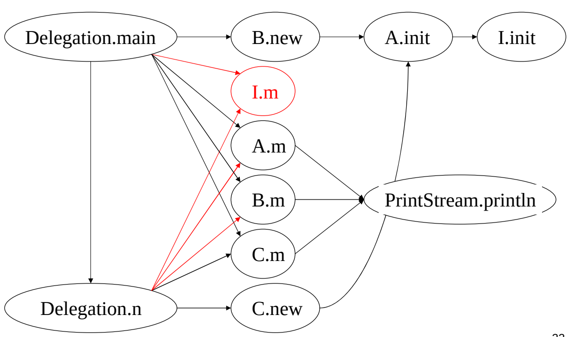
- Refinement of RA

- $Main.main \in R$
- $M.m \in R$
  - $e.n$ 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 $type(e)$
  - $\Rightarrow \forall\ N \in subtype(\ type(e)) : N.n \in R \wedge (M.m, N.n) \in E$

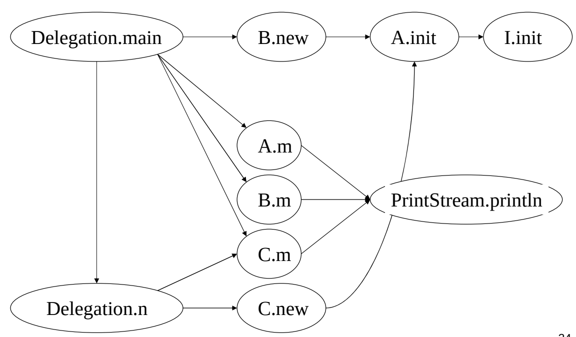
Example

```
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



CHA on Example



Rapid Type Analysis – RTA

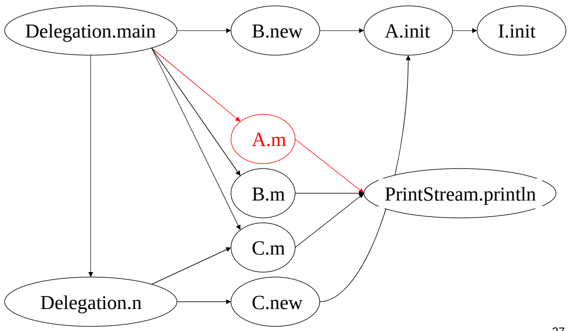
- Still simple and fast refinement of CHA
- Maintains reachable methods R and instantiated classes S
- Fixed point iteration: whenever S changes, we revisit the worklist R

- $Main.main \in R$
- For all class (static) methods s : $class(s) \in S$
- $M.m \in R$
  - $new\ N$ is a constructor call site in $M.m$
    $\Rightarrow N \in S \wedge N.new \in R \wedge (M.m, N.new) \in E$
  - $e.n$ is a field access / call site in $M.m$
    $\Rightarrow \forall\ N \in subtype(type(e)) \wedge N \in S: N.n \in R \wedge (M.m,N.n) \in E$

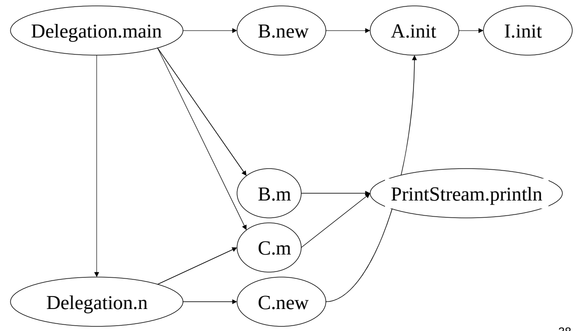
Example

```
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");}
}
```

RTA on Example



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() {
   A a = new A();
   a.n();
   sub();
}
sub(){
   A a = new B();
   a.n();
}
```

```
class A {
  public void n(){…}
}

class B extends A
  public void n(){…}
}
```


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

```
main() {
   A a = new A();
   X.dispatch(a);
   sub();
}
sub(){
   A a = new B();
   X.dispatch(a);
}
class X {
  public static void dispatch(A a){ a.n() }
}
```

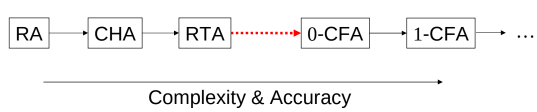
```
class A {
  public void n(){…}
}

class B extends A
  public void n(){…}
}
```


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



- 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

43

Notations

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

44

Separated Type Analysis – XTA

- Separate type sets S_m reaching methods m and fields x (treat fields x like methods pairs set_x, get_x)

- $Main.main \in R$
- $M.m \in R$
  - For all class (static) methods $s : class(s) \in S_{M.m}$
  - $new\ N$ is a constructor call site in $M.m$
    $\Rightarrow \quad N \in S_{M.m} \wedge N.new \in R \wedge (M.m, N.new) \in E$
  - $e.n$ is a field access / call site in $M.m$
    $\Rightarrow \forall\ N \in subtype(\ type(e)\ ) \wedge N \in S_{M.m} : N.n \in R \wedge$
    $subtype(\ param(\ N.n\ )\ ) \cap S_{M.m} \subseteq S_{N.n} \quad \wedge$
    $subtype(\ result(\ N.n\ )\ ) \cap S_{N.n} \subseteq S_{M.m} \quad \wedge$
    $(M.m, N.n) \in E$

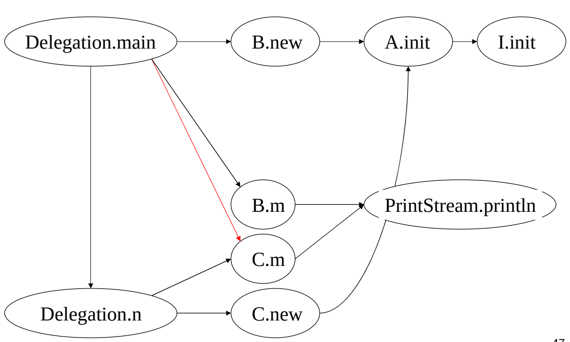
45

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");}
}
```

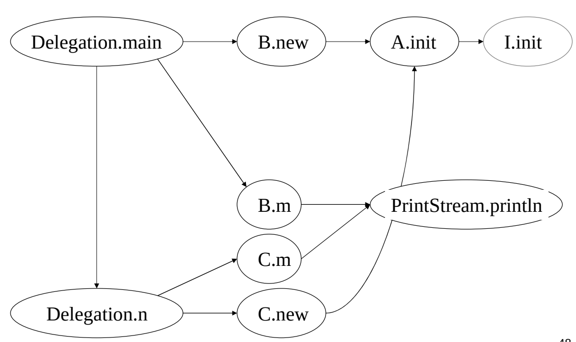
46

XTA on Example



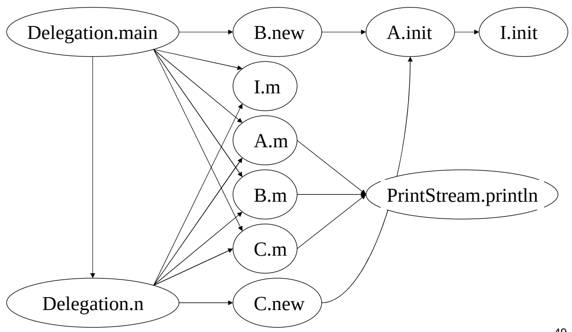
47

XTA on Example



48

RA vs XTA on Example



49

Increasing complexity

RA → CHA → RTA → XTA → 0-CFA → 1-CFA → …

- 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

50

Increasing precision

RA → CHA → RTA → XTA → 0-CFA → 1-CFA → …

- 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

51

Source

- Frank Tip and Jens Palsberg:
  Scalable Propagation-Based Call Graph Construction Algorithms.
  ACM Conf. on Object-Oriented Programming Systems, Languages and Application – OOPSLA 2000.
- David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers:
  Call Graph Construction in Object-Oriented Languages.
  OOPSLA 1997.

52

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)

53

Outline

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

54

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:

```
     A a,b,c;
     X x,y;
s1: a = new A( ) ; // Pt ( a ) = {o1}
s2: b = new A( ) ; // Pt ( b ) = {o2}
     b = a; // Pt ( b ) = {o1 , o2}
     c = b; // Pt ( c ) = {o1 , o2}
```

- Here `oi` means the object created at allocation site `si`.
- 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

56

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

```
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.
- Finitely many abstract objects

58

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:

```
          l = r          ⇒ Pt(r) ⊆ Pt(l)        (Assignment)
site i:   l = new A()    ⇒ {oi} ⊆ Pt(l)         (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 equations 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

59

Example

A Simple Program	Generated set constraints
`public A methodX(A param){`	
`      A a1 = param;`	`1: Pt(param) ⊆ Pt(a1)`
`s1 :  A a2 = new A( ) ;`	`2: o1 ∈ Pt(a2)`
`      A a3 = a1;`	`3: Pt(a1) ⊆ Pt(a3)`
`      a3 = a2 ;`	`4: Pt(a2) ⊆ Pt(a3)`
`      return a3 ;`	
`}`	

60

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

```
          l = r        ⇒ Pt(r) ⊆ Pt(l)       ⇒ r ⟶ l (Assignment)
site i:   l = new A()  ⇒ {oi} ⊆ Pt(l) ⇒ oi ⟶ l (Allocation)
```

- Previous example revisited

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

- P2G is our data-flow graph and the objects are our data values to be propagated.
- P2G initialization: ∀ `oi` ⟶ `l`, add `oi` to `Pt(l)`.
- P2G propagation: ∀ `r` ⟶ `l`, add let `Pt(l)=Pt(l)∪Pt(r)`

61

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(a)={o1,o2}
           //sensitive: Pt(a)={o1}
(4)        b = f
           //insensitive: Pt(b)={o1,o2}
           //sensitive: Pt(b)={o2}
```

- Our approach would have generated the following *set* of constraints `o1∈Pt(f), Pt(f)⊆Pt(a), o2∈Pt(f), Pt(f)⊆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 our 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 (SSA) representation

62

Method Calls: Object Transfer

```
s1 :   A a = new A() // o1 -> a
s2 :   X x1 = new X() // o2 -> x1
       a.storeX( x1 ) // x1 -> x4 , x4 -> x3 , x3 -> f
       X x2 = a.loadX() // f -> x2
classA {
   X f ;
   private void setX (X x3) {f = x3;}
   private X getX() {return f;}
   public void storeX (X x4) { this.setX(x4);}
   public X loadX() {return this.getX();}
}
```

- Involved object transport
  - Argument passing, i.e., assigning arguments to parameters (e.g. x1⟶x4).
  - A call `a.m()` involves an implicit assignment a⟶$\texttt{this}_m$.
  - The return assignment involves a implicit step too f⟶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

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

Procedural Definition

```
mName(A this,
        P1 p1, P2 p2,
        R ret) {
        …
        ret = Rexpr ;
}
```

OO Invocation

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

Procedural Invocation

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

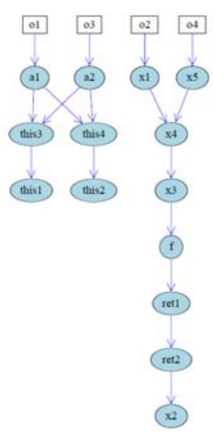

Advantage of Procedural Representation

- Given a call site `l=r0.m(r1,…,rn)`
  - Represented as `m(r0,r1,…,rn,l)`
  - Targeted at method `public R mName(P1 p1,P2 p2)` in `classA`
  - Represented as `m(A this,P1 p1,…,Pn pn,R ret)`
- We add the following P2G edges
- `r0⟶this, r1⟶p1, …, rn⟶pn,ret⟶l`
- Thus, each resolved call site results in a well-defined set of inter-procedural P2G edges.

Previous Example Revisited / Extended

```
class Main {
   static procedure main (Main this , String[] args) {
   s1 : A a1 = new A( ) ;      // o1 --> a1
   s2 : X x1 = new X( ) ;      // o2 --> x1
        storeX ( a1 , x1 ) ;   // a1 --> this3 , x1 --> x4
        X x2;
        loadX ( a1 , x2 ) ;    // a1 --> this4 , ret2 --> x2
   s3 : A a2 = new A( ) ;      // o3 --> a2
   s4 : X x5 = new X( ) ;      // o4 --> x5
        storeX ( a2 , x5 ) ;   // a2 --> this3 , x5 --> x4
        loadX ( a2 , x2 ) ;    // a2 --> this4 , ret2 --> x2
   }
}
classA {
   X f ;
   procedure setX(A this1, X x3 ) { f = x3 }          // x3 --> f
   procedure getX(A this2, Xret1 ) { ret1 = f }       // f --> ret1
   procedure storeX(A this3, X x4 ) { setX(this3,x4) }
        // this3 --> this1, x4 --> x3
   procedure loadX(A this4 , Xret2) {getX( this4, ret2 ) }
        // this4 --> this2 , ret1 --> ret2
}
```


P2G Generated



Resolving Call Targets

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

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 $T_1.m()$, … $T_n.m()$.
- 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*().

69

Static Dispatch

- Given a conservative call graph we can construct a function staticDispatch(`a.m()`) that provides us with a set of possible target methods for any given call site `a.m()`.
- We can then proceed as follows:

```
for each call site l = r0.m(r1,…,rn) do
    let targets = staticDispatch(r0.m(…))
    for each method m(A this,P1 p1,…,Pn pn,R ret) ∈ targets do
        add P2G edges r0->this, r1->p1, … , rn->pn, ret->l
```

- 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.
- 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.

70

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()` with signature *m* on an object O_A of type *A*
- We can then proceed as follows:

```
for each call site l = r0.m(r1,…, rn) (or m(r0,r1,…,rn,l)) do
    for each object O_A ∈ Pt(r0) do
        1. Let m = signatureOf(m())
        2. Let A = typeOf(O_A)
        3. Let m(A this,P1 p1,…,Pn pn, R ret)=
               dynamicDispatch(A,m)
        4. Add P2G edges r0->this, r1->p1, …, rn->pn, ret->l
```

- 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.

71

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)=∅                  //O(#v)
2. For each allocation edge oi->v do                //O(#o)
    (a) let Pt(v)=Pt(v) ∪ {oi}
    (b) add v to worklist
3. Repeat until worklist empty                      //O(#v*#o)
    (a) Remove first node p from worklist
    (b) For each edge p->q do                       //O(#v)
        i. Let Pt(q)=Pt(q) ∪ Pt(p)
        ii.If Pt(q) has changed, add q to worklist
```

- Time complexity: Let $\#v$ be the number of variable nodes and $\#o$ 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 $\#o$ times for each node, thus, we have $O(\#v*\#o)$ number of work list iterations.
- Each such iterations may update every other variable node (hence $O(\#v)$ within the loop). Thus, an upper limit is $O(\#v^2*\#o)$.

72

Optimizing the Analysis

- The high time complexity $O(\#v^2*\#o)$ encourages optimizations. Optimizations can basically be done in two different ways:
- We can reduce the size of P2G by identifying points-to sets that must be equal. This idea will be exploited in
  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:
  3. Topological node ordering.
- Other optimizations are possible all three are simple and effective.

73

Previous Example Revisited: Results of Points-to Analysis

```
c l a s s Main {
    s t a t i c procedure main (Main this , St r i n g [ ] 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 ) ;
    }
}
classA {
    X f ; // Pt ( f ) = {o2 , o4}
    procedure setX (A thi s1 , X x3 ) { f = x3 ;} // Pt ( t h i s 1 ) = {o1 , o3 } , Pt ( x3 ) = {o2 , o4}
    procedure getX(A thi s2 , X r1 ) { r1 = f ;}  // Pt ( t h i s 2 ) = {o1 , o3 } , Pt ( r1 ) = {o2 , o4}
    procedure storeX (A thi s3 , X x4 ) { setX ( thi s3 , x4 ) ;}
           // Pt ( t h i s 3 ) = {o1 , o3 } , Pt ( x4 ) = {o2 , o4}
    procedure loadX (A thi s4 , X r2 ) {getX( thi s4 , r2 ) ;}
           // Pt ( t h i s 4 ) = {o1 , o3 } , Pt ( r2 ) = {o2 , o4}
}
```

74

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(o3.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 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-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.

75

Outline

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

76