Inter-Procedural Analysis and Points-to Analysis

Welf Löwe
Welf.Lowe@lnu.se

Outline

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

2

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

3

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).

4

Proper Call Graphs

- A proper call graph is in addition
  - Conservative: Every call `A.m()` → `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 an Event Listener interface will have the Event Handler method as an additional entry point if we are neglecting the Event Generator 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.

5

Techniques for Inter-Procedural Analysis

- Intra-procedural analysis on call and basic block graphs
- Simulated execution

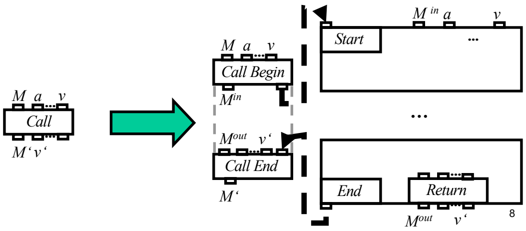
6

Call and basic block graphs

- Given call graph and a bunch of procedures each with a basic block graph
- Merge call 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 for intra-procedural analysis

Merging call and basic block graphs

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

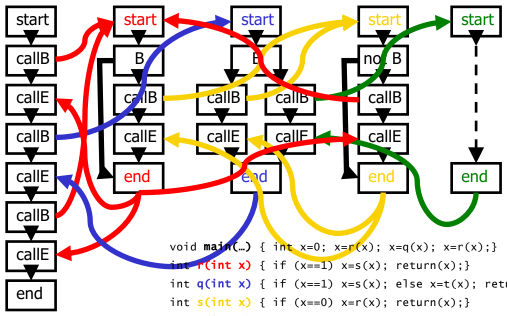


Example Program

```
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

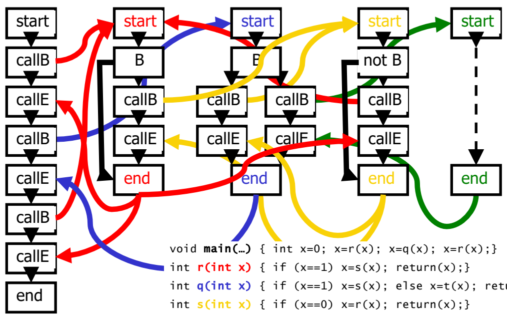



```
void main(…) { int x=0; x=r(x); x=q(x); x=r(x);}
int r(int x) { if (x==1) x=s(x); return(x);}
int q(int x) { if (x==1) x=s(x); else x=t(x); return(x);}
int s(int x) { if (x==0) x=r(x); return(x);}
int t(int x) { return(x+1);}
```


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)
- Context-sensitive analysis mitigates this problem

Example: Unrealizable Path




```
void main(…) { int x=0; x=r(x); x=q(x); x=r(x);}
int r(int x) { if (x==1) x=s(x); return(x);}
int q(int x) { if (x==1) x=s(x); else x=t(x); return(x);}
int s(int x) { if (x==0) x=r(x); return(x);}
int t(int x) { return(x+1);}
```

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
- Points to analysis
- Points to analysis (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 are we interested?

- Elimination of dead code i.e., classes never loaded, no objects created from, and methods never called.
- Elimination of polymorphism: usage refers to a statically known method i.e., only one target is possible.
- 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, and Points-to analysis.
- Detection of design patterns (e.g., singletons usage refers to a single object, not to a set of objects of the same type) and anti-patterns.
- Architecture recovery i.e. the reconstruction of a system architecture from code

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 call `a.m()` occurring in a method definition `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).
- Class objects (static objects) are treated as objects

19

Declared Sources

- Stack objects are considered part of *this*
  - Let a be a local variable or parameter, resp.
  - $a.m()$ is a usage of whatever a is declared as (target),
  - in whatever *this* is declared as (source).

20

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) $\Leftrightarrow (Class(a).x, Class(b).y) \in E$
- 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) $\Leftrightarrow (N(a).x, N(b).y) \in E$
- A name schema N a is an abstraction function with finite co-domain
- The declared $Class(o)$ is a special name schema and, hence, describes a special call graph

21

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)
  - …

22

Decidability of a Call Graph

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

23

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 $b.y \Leftarrow (N(a).x, N(b).y) \in E$

24

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

Algorithms to discuss

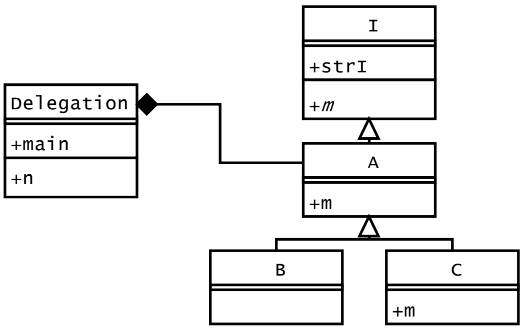
All algorithms these are conservative:

- 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 *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 $\Rightarrow \forall N \in Program$: $N.n \in R \wedge (M.m, N.n) \in E$

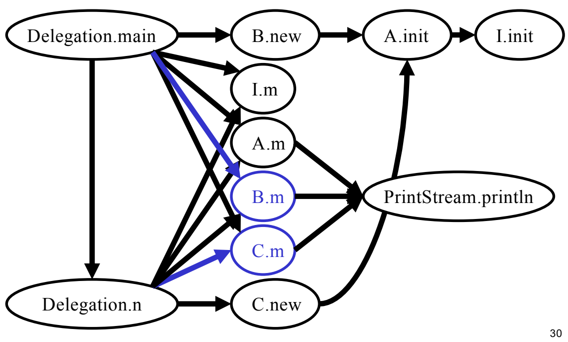
Example



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();}
   }
abstract class I {
   public String strI = "Printing I string";
   public void m();
}
class A extends 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");}
}
```


RA on Example



Class Hierarchy Analysis – CHA

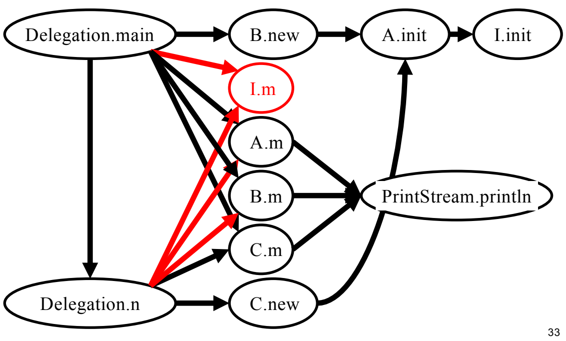
- Refinement of RA
- $Main.main \in R$
- $M.m \in R$
  - $e.n$ is a field access / call site in $M.m$
  - $type(e)$ is the static (declared) type of access path expression e
  - $subtype(type(e))$ is the set of (declared) sub-types of $type(e)$

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

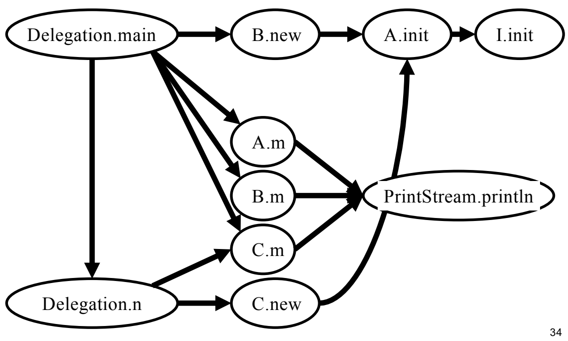
Example

```
public class Delegation {
   public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();}
   public static void n() {
        new C().m();}
}
abstract class I {
   public String strI = "Printing I string";
   public void m();
}
class A extends 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 $M.m$

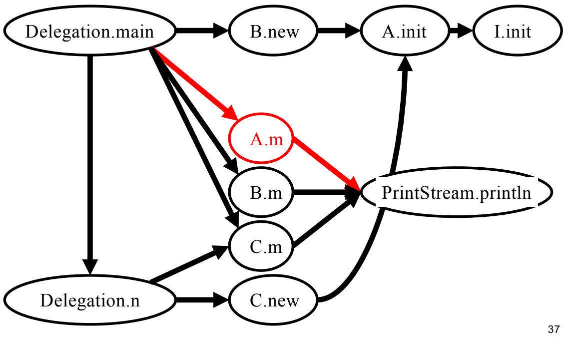
    $\Rightarrow N \in S \wedge N.new \in R \wedge (M.m, N.new) \in E$
  - $e.n$ is a field access / call site in $M.m$

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

Example

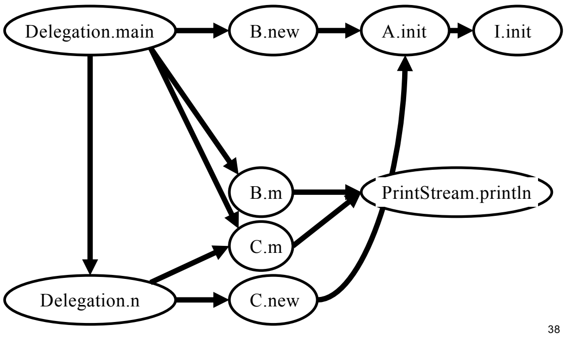
```
public class Delegation {
   public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();}
   public static void n() {
        new C().m();}
}
abstract class I {
   public String strI = "Printing I string";
   public void m();
}
class A extends I {
   public 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



37

RTA on Example



38

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(){…}
}
```

39

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(){…}
}
```

40

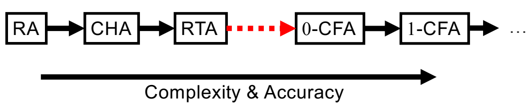
Control Flow Analysis

- 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

41

Order on Algorithms

- Increasing complexity
- Increasing accuracy



- Analyses between RTA and 0-CFA?

42

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

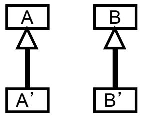
Example

```
Class M {
    // S_M.m = {A, A', B', N_static}
    m(){
        A a; B b;
        if (?)
            a=new A'();
        else
            a=new A();
        b=N.n(a); ...
    }
}
```

$// \ S_{M.m} = \{A, A', B', N_{static}\}$

```
Class N {
    // S_N.n = {A', B, B', N_static}
    static n(A' a):B' {
        new B(); ...
        return new B'();
    }
}
```

$// \ S_{N.n} = \{A', B, B', N_{static}\}$



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

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 $M.m$

    $\Rightarrow \forall\ N \in subtype(\ type(e)\ ) \wedge N \in S_{M.m} : N.n \in R \wedge$

    $subtype(\ param(\ N.n\ )\ ) \cap S_{M.m} \subseteq S_{N.n} \quad \wedge$

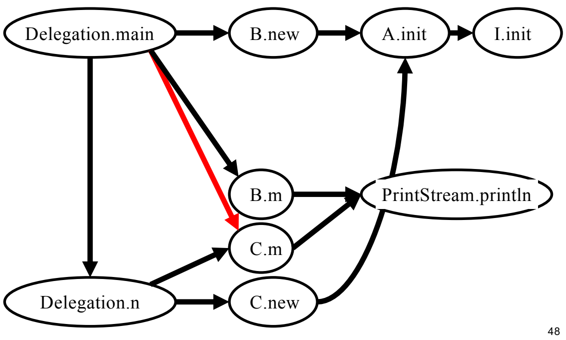
    $subtype(\ result(\ N.n\ )\ ) \cap S_{N.n} \subseteq S_{M.m} \quad \wedge$

    $(M.m, N.n) \in E$

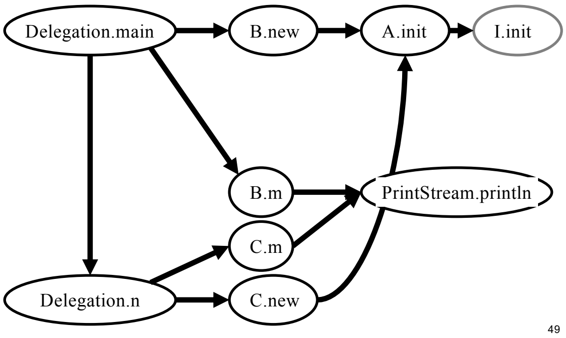
Example

```
public class Delegation {
    public static void main(String args[]) {
          A i = new B();
          i.m();
          Delegation.n();}
    public static void n() {
          new C().m();}
}
abstract class I {
    public String strI = "Printing I string";
    public void m();
}
class A extends 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

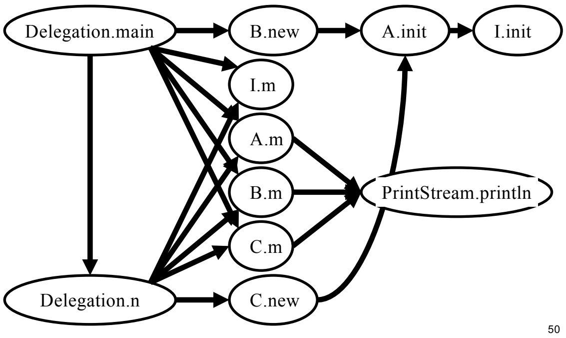


XTA on Example



49

RA vs XTA on Example



50

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

51

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

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
- Points to analysis
- Points to analysis (fast and precise, not today – requires SSA)

54

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.

55

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

Outline of the 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.

57

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`	$\Rightarrow Pt(r) \subseteq Pt(l)$	(Assignment)
`site i:`	`l = new A()`	$\Rightarrow \{oi\} \subseteq Pt(l)$	(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

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

Generated set constraints

1: $Pt(param) \subseteq Pt(a1)$
2: $o1 \in Pt(a2)$
3: $Pt(a1) \subseteq Pt(a3)$
4: $Pt(a2) \subseteq Pt(a3)$

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 abstract objects are our data values to be propagated.
- P2G initialization: ∀`oi→l`, let `Pt(l)=Pt(l)∪{oi}`
- P2G propagation: ∀`r→l`, let `Pt(l)=Pt(l)∪Pt(r)`

Assignments and Allocations
(flow-insensitive vs. flow-sensitive in a method)

- Recall Assignment and Allocation
  - Constraints: `Pt(r) ⊆ Pt(l)` and `oi ∈ Pt(l)`, resp.
  - Partial graph generated: `r→l` and `oi → l`, resp.

```
(1) s1: f = new A()
(2)     a = f
(3) s2: f = new A()
        //insensitive: Pt(a)={o1,o2}
        //sensitive: Pt(a)={o2}
(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}` (at least) 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 required 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.

Representation of Methods

OO Definition	Procedural Definition
`class A {`	`m(A this,`
`public R m(P1 p1,P2 p2){`	`P1 p1, P2 p2,`
`…`	`R res) {`
`return Rexpr;`	`…`
`}`	`res = Rexpr ;`
	`}`
OO Invocation	**Procedural Invocation**
`l = a.m(x,y);`	`m(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 m(P1 p1,P2 p2)` in `classA`
  - Represented as `m(A this,P1 p1,…,Pn pn,R res)`
- For calls, 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.

Method Calls and Definitions
(flow-sensitive between methods)

- Method Calls $l = m(r_0, r_1, r_2, \ldots)$ and Definitions $m(this, p_1, p_2, \ldots, res)\{ \ldots \}$
  - Constraints $Pt(r_0) \subseteq Pt(this_{A.m})$, $Pt(r_i) \subseteq Pt(p_i)$, $Pt(res_{A.m}) \subseteq Pt(l)$
  - Partial graph $r0 \rightarrow this_{A.m}$, $r1 \rightarrow p1, \ldots, rn \rightarrow pn$, $res_{A.m} \rightarrow l$

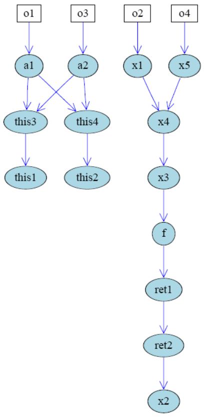
```
s1: A a = new A()     // o1→a
s2: X x1 = new X()    // o2→x1
    a.storeX( x1 )    // a→this3 x1→x4
    X x2 = a.loadX()  // res4→x2 a→this4
Class A {
   X f ;
   m1: private void setX (X x3) {f = x3;} x3→f
   m2: private X getX() {return f;} f→res2
   m3: public void storeX (X x4) {this.setX(x4);} this3→this1 x4→x3
   m4: public X loadX() {return this.getX();} this4→this2 res2→res4
}
```

- Involved object transport
  - Argument passing, i.e., assigning arguments to parameters (e.g. `x1→x4`).
  - A call `a.m()` involves an implicit assignment `a→this`.
  - The return assignment involves implicit steps `f→ … → x2` .

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



67

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 find the target methods.
- Recall from previous lecture:
  - 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.

68

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 $l\ =\ r0.m(r1,\dots,rn)$ do

  let targets = staticDispatch($r0.m(\dots)$)

  for each method $m(A\ this, P1\ p1,\dots,Pn\ pn, R\ res) \in$ targets do

  add P2G edges $r0\rightarrow this$, $r1\rightarrow p1$, … , $rn\rightarrow pn$, $res\rightarrow l$
- 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.

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 $l\ =\ r0.m(r1,\dots,\ rn)$ (or $m(r0,r1,\dots,rn,l)$) do

  for each abstract object $o_A \in Pt(r0)$ do

  1. Let m = signatureOf($m()$)
  2. Let A = typeOf(o_A)
  3. Let $m(A\ this, P1\ p1,\dots,Pn\ pn,\ R\ res)$ = dynamicDispatch(A, m)
  4. Add P2G edges $r0\rightarrow this$, $r1\rightarrow p1$, …, $rn\rightarrow pn$, $res\rightarrow l$
- 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

DFA on a Complete P2G

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

  1. For each variable v let $Pt(v)=\emptyset$ //$O(\#v)$
  2. For each allocation edge $oi\rightarrow v$ do //$O(\#o)$
     (a) let $Pt(v)=Pt(v)\cup\{oi\}$
     (b) add v to worklist
  3. Repeat until working list empty //$O(\#v*\#o)$
     (a) Remove first node p from worklist
     (b) For each edge $p\rightarrow q$ do //$O(\#v)$
        i. Let $Pt(q)=Pt(q)\cup Pt(p)$
        ii. If $Pt(q)$ has changed, add q to working list
- Time complexity: Let $\#v$ be the number of variable nodes and $\#o$ the number of (abstract) objects.
- A variable 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.

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 ( x2 ) = {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 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(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 abstract object have its own set of fields. This would however correspond to object/method inlining and increase the number of P2G nodes 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
- Points to analysis
- Points to analysis (fast and precise, not today – requires SSA)