Linnéuniversitetet
Kalmar Växjö

Inter-Procedural Analysis and Points-to Analysis

Welf Löwe
Welf.Lowe@lnu.se

1

Outline

Part 1: Data Flow Analysis and Abstract Interpretation
Part 2: Inter-procedural and Points-to analysis
Part 3: Static Single Assignment (SSA) form
Part 4: SSA based optimizations

2

2

Outline Part 2

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

3

3

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

4

4

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).

5

5

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.

6

6

Techniques for Inter-Procedural Analysis

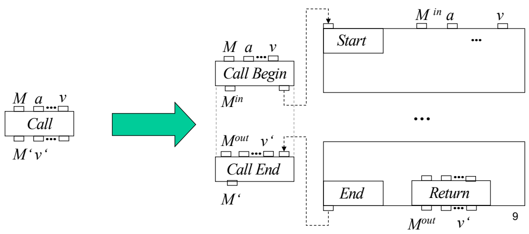
- Data structure used
  - Call graphs encoding the calls between the methods and
  - Basic block graphs or SSA graphs encoding the procedures/methods
- Analysis technique
  - Inter-procedural DFA or
  - Simulated execution

Call and basic block graphs

- Given call graph and a bunch of procedures/methods each with a basic block graph
- Idea for inter-procedural DFA: merge call and 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) data flow analysis
- Polymorphism is resolved by explicit dispatcher or by several targets'
- Inter-procedural data flow analysis now (technically) 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

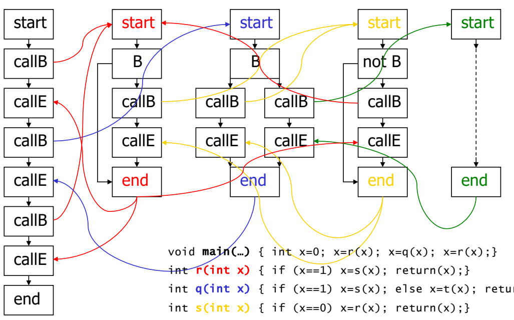


Example Program

```
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

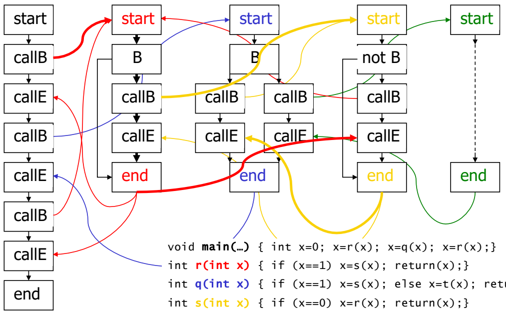



```
void main(…) { int x=0; x=r(x); x=q(x); x=r(x);}
int r(int x) { if (x==1) x=s(x); return(x);}
int q(int x) { if (x==1) x=s(x); else x=t(x); return(x);}
int s(int x) { if (x==0) x=r(x); return(x);}
int t(int x) { return(x+1);}
```


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 (too) conservative
- Still correct (and still, in general, more precise than corresponding intra-procedural analyses)
- Call-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);}
```

13

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 …

14

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 the theoretical frameworks of monotone DFA and Abstract Interpretation

15

Outline

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

16

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
  - Class loading

17

Why are we interested?

- 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.
- 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.
- 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

18

Call Graphs: The Basic Problem

- The difficult task of any call (member) graph construction algorithm is to approximate the set of methods (members) that can be targeted at different call sites (member reference points).
  - 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.

19

19

Declared Target

- Simple call graphs can be calculated based on the declared targets of calls.
- 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 for call graph construction, 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

20

20

Generalized Call Graphs

- A simple 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) ⇔ $(Class(a).x, Class(b).y) \in E$
- A 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 is an abstraction function with a finite co-domain
- The $Class(o)$ is a special name schema and, hence, describes a special type of call graphs

22

22

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)
  - …

23

23

Simplification: $N(o)=Class(o)$

- For a first try, we consider only one name schema:
  - Distinguish objects of different classes / types
  - Formally, $N(o)=Class(o)$
- Consequently, all these call graphs are …
  - a directed graphs $G=(V,E)$
  - 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 a method / constructor / field y instance of B) ⇔ $(A.x , B.y) \in E$
- Not decidable, we need to find optimistic and conservative approximations

24

24

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 you could decide the exact call graph, you knew if the program terminates or not
- Decidable if name schema is abstract enough (but then not relevant in practice)

25

25

Approximations

- Simple call graph constitutes a conservative approximation
  - from static semantic analysis
  - declared class references in a class A and their subtypes are potentially used in A
  - $a.x$ really uses $b.y \Rightarrow (Class(a).x, Class(b).y) \in E$
- 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 $b.y \Leftarrow (N(a).x, N(b).y) \in E$

Algorithms to discuss

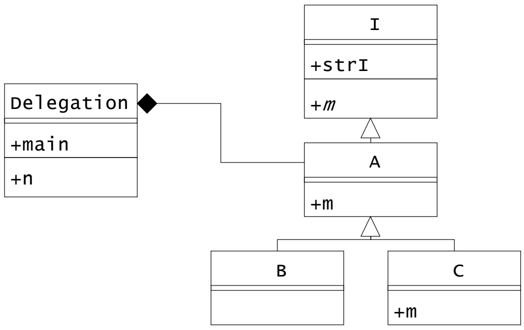
All algorithms these are conservative:

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

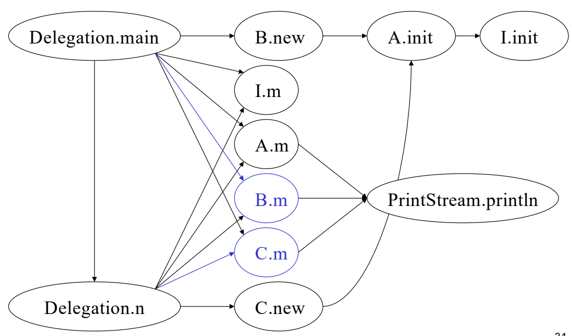
Example



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

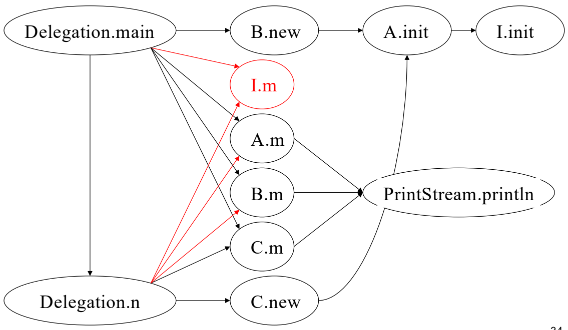
- $Main.main \in R$
- $M.m \in R$
  - $e.n$ is a field access / call site in $M.m$
  - $type(e)$ is the static (declared) type of access path expression e
  - $subtype(type(e))$ is the set of (declared) sub-types of $type(e)$

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

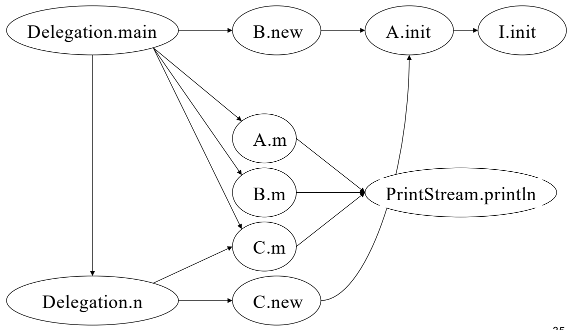
Example

```
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 $M.m$

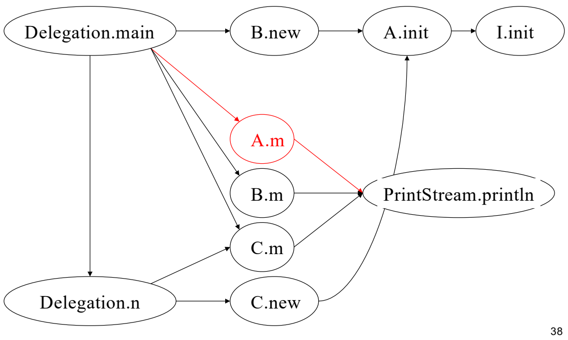
    $\Rightarrow N \in S \wedge N.new \in R \wedge (M.m, N.new) \in E$
  - $e.n$ is a field access / call site in $M.m$

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

Example

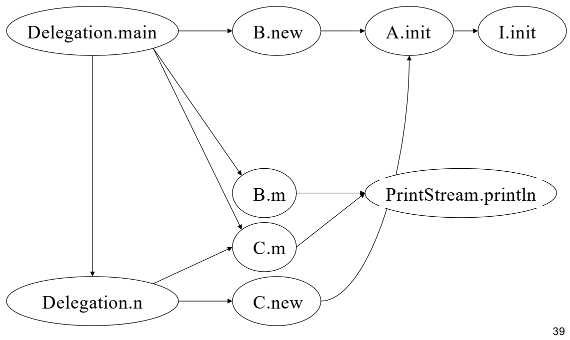
```
public class Delegation {
   public static void main(String args[]) {
        A i = new B();
        i.m();
        Delegation.n();}
   public static void n() {
        new C().m();}
}
abstract class I {
   public String strI = "Printing I string";
   public void m();
}
class A extends I {
   public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



38

RTA on Example



39

Context-Insensitive Control Flow Analysis – 0-CFA

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

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


40

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


41

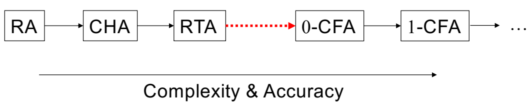
Control Flow Analysis

- 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 idea:
    - Make k adaptive over the analysis
    - Focus with large k on specific program parts
    - Reduce k to min if analysis time / space not sufficient or if different contexts give the same result

42

Order on Algorithms

- Increasing complexity
- Increasing accuracy



- Analyses between RTA and 0-CFA?

43

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
- Requires fixed point iteration

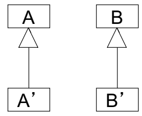
44

44

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); ...
  }
}
```

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



45

45

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

46

46

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 $M.m$
    $\Rightarrow\ N \in S_{M.m} \wedge N.new \in R \wedge (M.m, N.new) \in E$
  - $e.n$ is a field access / call site in $M.m$
    $\Rightarrow\ \forall\ N \in subtype(type(e)) \wedge N \in S_{M.m} : N.n \in R\ \wedge$
    $subtype(param(N.n)) \cap S_{M.m} \subseteq S_{N.n}\ \wedge$
    $subtype(result(N.n)) \cap S_{N.n} \subseteq S_{M.m}\ \wedge$
    $(M.m, N.n) \in E$

47

47

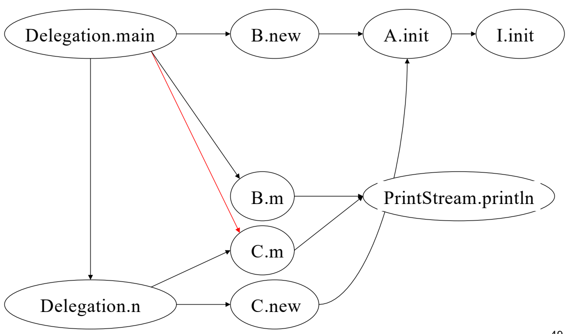
Example

```
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");}
}
```

48

48

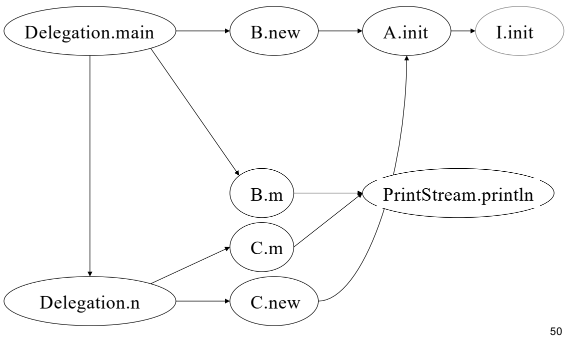
XTA on Example



49

49

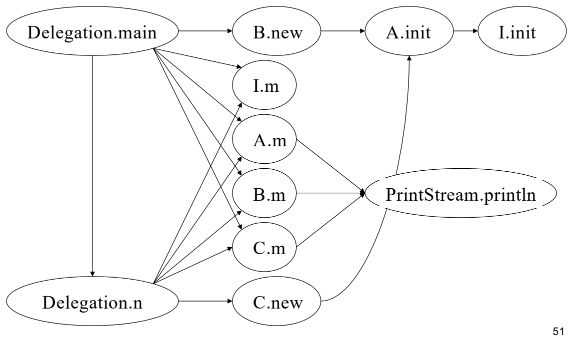
XTA on Example



50

50

RA vs XTA on Example



51

51

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

52

52

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

53

53

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)

54

54

Outline

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

55

55

Client-Applications of Points-to Analysis

- Points-to results can be used as input for several 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.

56

56

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

57

57

Outline of the approach

Points-to analysis (as any DFA) 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.

58

58

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` depended on some input values).
- From now on, each object creation point (`new A(), a.clone(), "hello"`) represents a unique abstract object (identified by the source code location).
- Replaces the simple declared-class-based name schema
- Again, many run-time objects are mapped to a single abstract object.
- Finitely many abstract objects

59

59

Object Transport as Set Constraints

- Abstract 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:

	`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

60

60

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: $Pt(a2) \subseteq Pt(a3)$

61

61

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 (allocations): ∀oi→l, let Pt(l)=Pt(l)∪{oi}
- P2G propagation (assignments): ∀r→l, let Pt(l)=Pt(l)∪Pt(r)

62

62

Flow-insensitive vs. flow-sensitive analysis
(within a methods)

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

- Our approach would have generated the following 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.

63

63

Representation of Methods

OO Definition

```
class A {
public R m(P1 p1,P2 p2){
  …
  return Rexpr;
}
```

Procedural Definition

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

OO Invocation

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

Procedural Invocation

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

64

64

Uniformly using the Procedural Representation

- Given a call site *l=r0.m(r1,…,rn)*
  Represented as *m(r0,r1,…,rn,l)*
- Targeted at method R m(P1 p1,P2 p2) defined in class *A*
  Represented as *m(A this,P1 p1,…,Pn pn,R res)*
- We add the following P2G edges
- *r0→this, r1→p1, …, rn→pn,ret→l*
- Each resolved call site results in a well-defined set of inter-procedural P2G edges.

65

65

Method Calls and Definitions
(always flow-sensitive between methods)

- Call site l = m(r0, r1, r2,…)
- Target method m(this, p1, p2,…, res){…} in *A*

- 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
  - $r_0 \rightarrow this_{A.m}$,
  - $r_1 \rightarrow p_1, \ldots, r_n \rightarrow p_n$,
  - $res_{A.m} \rightarrow l$
- Involved object transport
  - Argument passing, i.e., assigning arguments to parameters
  - A call a.m() involves an implicit assignment a → this
  - The return assignment res → l

66

66

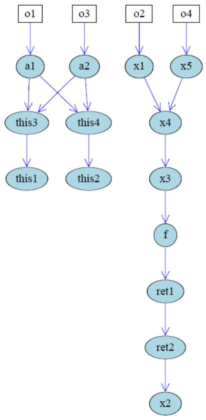
Previous Example Revisited / Extended

```
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
   }
}
class A {
   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, X ret2){getX(this4,ret2)}
        // this4 → this2, ret1 → ret2
}
```

67

67

P2G Generated



DFA on a P2G

- In this DFA implementation, 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 $O(\#v^2 * \#o)$.

Optimizing the Analysis

- The high time complexity $O(\#v^2 * \#o)$ encourages optimizations. Optimizations can basically be done in three different ways (all three simple and effective):
- 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 cleverer ordering:
  3. Topological node ordering.
- Other optimizations are possible too.

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 consider 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*().

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 $l = r0.m(r1, \ldots, rn)$ do
   let targets = staticDispatch($r0.m(\ldots)$)
   for each method $m(A\ this, P1\ p1, \ldots, Pn\ pn, R\ res) \in$ 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.
  - Then the P2G is *complete* as no more edges are to be added. Complete P2Gs are much easier to handle in the subsequent DFA phase, which only does object propagation.
- Disadvantage: The precision of the externally derived call graph influences the points-to-analysis.

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 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,…,Pn pn,R res)* = dynamicDispatch(A, *m*)
  4. Add P2G edges *r0→this, r1→p1, …, rn→pn, res→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 which requires a fixed point iteration

74

74

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(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);
   }
}
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}
}
```

75

75

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/sets 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.

76

76

Outline

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

77

77

Outline

Part 1: Data Flow Analysis and Abstract Interpretation

Part 2: Inter-procedural and Points-to analysis

Part 3: Static Single Assignment (SSA) form

Part 4: SSA based optimizations

78

78