OOPLs - call graph construction

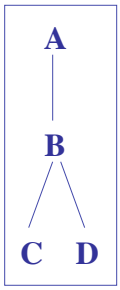
- Compile-time analysis of reference variables and fields
  - Problem: how to resolve virtual function calls?
    - Need to determine to which objects (or types of objects) a reference variable may refer during execution
  - Type hierarchy-based methods
    - Class hierarchy analysis (CHA)
    - Rapid type analysis (RTA)
  - Flow-based methods
    - Field-sensitive, flow-insensitive, context-insensitive reference (i.e., points-to) analysis

Example – executed calls

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



Reference Analysis

- OOPLs need type information about objects to which reference variables can point to resolve dynamic dispatch
- Often data accesses are indirect to object fields through a reference, so that the set of objects that might be accessed depends on execution-time values of reference variables
- Need to pose this as a compile-time program analysis with representations for reference variables/fields, objects and classes.

Reference Analysis

- Different algorithms and program representation choices affect precision and cost
  - Class analyses use an abstract object (with or without fields) to represent all objects of a class
  - Points-to analyses use object instantiations, grouped by some mechanism (e.g., creation sites)
- The analysis can incorporate information about flow of control in the program or ignore it
  - *Flow sensitivity* (accounts for statement order)
  - *Context sensitivity* (separates calling contexts)

Reference Analysis

- Program representation used for analysis can incorporate reachability of methods as part of the analysis or can assume all methods are reachable
- Techniques can be differentiated by their solution formulation (that is, kinds of relations:
  - e.g., for reference assignments
    p = q, interpreted as
    Pts-to(q) ⊆ Pts-to(p) vs. Pts-to(q) = Pts-to(p)

Class Hierarchy Analysis

- Earliest method for reference analysis was CHA by Craig Chamber's group (UWashington)
  - Idea: look at class hierarchy to determine what classes of object can be pointed to by a reference declared to be of class A,
    - in Java this is the subtree in inheritance hierarchy rooted at A, *cone (A)*
  - Makes assumption that whole program is available and that all methods are reachable
  - Ignores flow of control
  - Uses 1 abstract object per class
  - Cheap, very approximate. safe

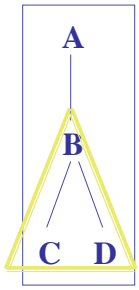
J. Dean, D. Grove, C. Chambers, ***Optimization of OO Programs Using Static Class Hierarchy,*** **ECOOP' 95**

CHA Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```

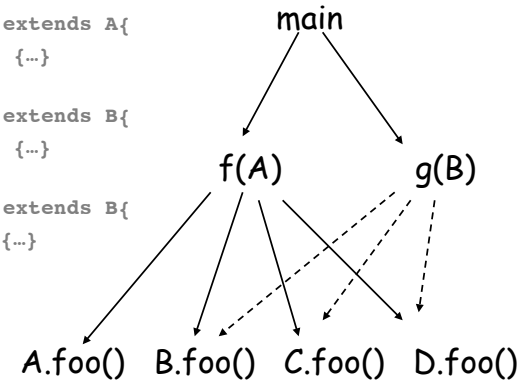


Cone(Declared_type(receiver))

CHA Example

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



Call Graph

More on CHA

- Type of receiver needn't be uniquely resolvable to de-virtualize a call
  - Need *applies-to* set for each method (the set of classes for which this method is the target when the run-time type of the receiver is one of those classes)
    - At a call site, take set of possible classes for receiver and intersect that with each possible method's applies-to set.
    - If only one method's set has a non-empty intersection, then invoke that method directly
    - Otherwise, need to use dynamic dispatch at runtime
  - Also can use run-time checks of actual receiver type (through reflection) to cascade through a small number of choices for direct calls, given predictions due to static or dynamic analysis

Rapid Type Analysis

- Improves CHA
- Constructs call graph on-the-fly, interleaved with the analysis
- Key idea: only expands calls if has seen an instantiated object of appropriate type
  - Ignores classes which have not been instantiated as possible receiver types
- Makes assumption that whole program is available and that all methods are reachable
- Uses 1 abstract object per class

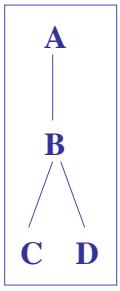
D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls", OOPSLA'96

RTA Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

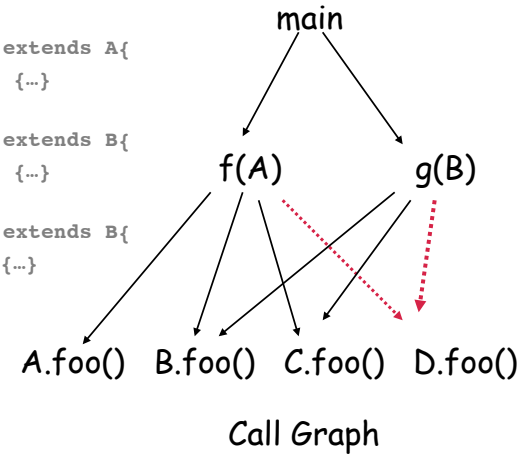
```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



RTA Example

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



Call Graph

Comparisons

Bacon-Sweeney, OOPSLA'96

```
class A {
public :
  virtual int foo(){ return 1; };
};
class B: public A {
public :
  virtual int foo(){ return 2; };
  virtual int foo(int i) { return i+1; };
};
void main() {
   B* p = new B;
   int result1 = p->foo(1);
   int result2 = p->foo( ) ;
   A* q = p;
   int result3 = q->foo( );
}
```

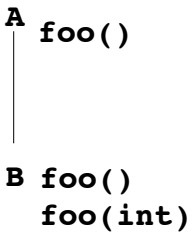
CHA resolves result2 call uniquely to B.foo() because B has no subclasses, however it cannot do the same for the result3 call.
RTA resolves the result3 call uniquely because only B has been instantiated.

Type Safety Limitations

Bacon-Sweeney, OOPSLA'96

- CHA and RTA both assume type safety of the code they examine

```
//#1
void* x = (void*) new B
B* q = (B*) x;//a safe downcast
int case1 = q->foo()
//#2
void* x = (void*) new A
B* q = (B*) x;//an unsafe downcast
int case2 = q->foo()//probably no error
//#3
void* x = (void*) new A
B* q = (B*) x;//an unsafe downcast
int case3 = q->foo(666)//runtime error
```



These analyses can't distinguish these 3 cases!

Experimental Comparison

Bacon and Sweeney, OOPSLA'96

Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	11,157	IDL specification to C++ stub-code translator
lcom	17,278	Compiler for the "L" hardware description language
hotwire	5,335	Scriptable graphical presentation builder
simulate	6,672	Simula-like simulation class library and example
idl	30,288	SunSoft IDL compiler with demo back end
taldict	11,854	Taligent dictionary benchmark
deltablue	1,250	Incremental dataflow constraint solver
richards	606	Simple operating system simulator

Table 1: Benchmark Programs. Size is given in non-blank lines of code

Data Characteristics

- Frequency of execution matters
  - Direct calls were 51% of static call sites but only 39% of dynamic calls
  - Virtual calls were 21% of static call sites but were 36% of dynamic calls
- Results they saw differed from previous studies of C++ virtuals
  - Importance of benchmarks
  - Paper was at a time when C++ programs were usually transformed C codes (didn't use virtual methods as much as modern codes)

Findings

- RTA was better than CHA on virtual function resolution, but not on reducing code size
  - Inference is that call graphs constructed have same node set but not same edge set!
- Claim both algorithms cost about the same because the dominant cost is traversing the cfg's of methods and identifying call sites, can pick up object creations during traversal
- Claim that RTA is good enough for call graph construction so that more precise analyses are not necessary for this task

Dimensions of Analysis

- How to achieve more precision in analysis for increased cost?
  - Incorporate flow in and out of methods
  - Refine abstract object representing a class to include its fields
  - Incorporate locality of reference usage in program into analysis rather than 1 'references' solution over the entire program
  - Always use reachability criteria in constructing call graph

Points-to Analysis for Java

- Points-to analysis traces flow of values through pointers (or reference variables and fields) in order to resolve virtual calls and trace side effects through indirect writes
- Historical roots in points-to analysis for C
  - Steensgaard's algorithm
  - Andersen's algorithm
  - Flow- and context sensitivity
- Field-sensitive analysis for Java
  - Based on Andersen for C augmented with handling for fields and dynamic dispatch

Flow & Context Sensitivity in Program Analysis

- Flow sensitivity
  - Analysis calculates a different solution at each program point
  - Analysis captures the sequential order of executions of statements
  - Expensive and highly accurate
- Context sensitivity
  - Analyze a method separately for different calling contexts (e.g., call sites)
  - Required often for accuracy for security and side effects clients

Points-to Analyses for C

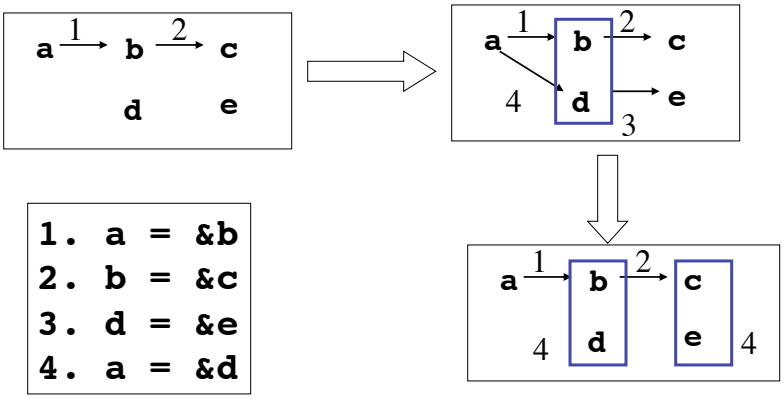
- Popular flow- and context-insensitive formulations of points-to analysis
  - Scalable to large codes (MLOC)
  - Good enough for ensuring safety of some optimizations
  - Good for program understanding applications
  - Not great for applications needing precise def-use information (e.g., program slicing, testing)
- Solution procedure utilizes *unification* or *inclusion* constraints
  - P = Q either implies PtsTo(P) = PtsTo(Q) or PtsTo(Q) $\subseteq$ PtsTo(P)
- Extended to points-to analyses for OOPL reference variables/fields

Points-to Analyses for C

- Bjarne Steensgaard's algorithm (POPL' 96)
  - Uses unification constraints so that for pointer assignments, p = q, algorithm makes PtsTo(p)=PtsTo(q)
    - This union operation is done recursively for multiple-level pointers
  - Reduces the size of the points-to graph (in terms of both nodes and edges)
    - *Almost linear* solution time in terms of program size, O(n) using fast union-find algorithm
    - Imprecision stems from merging points-to sets
  - One points-to set per pointer variable over entire program

cf M Shapiro and S. Horwitz, "Fast and Accurate Flow-insensitive Points-to Analysis" POPL'97

Steensgaard - Example




```
1. a = &b
2. b = &c
3. d = &e
4. a = &d
```

Points-to sets found:

PtsTo(a)={b,d}
PtsTo(b,d)={c,e}

Steensgaard Solution Procedure - At a glance

- Find all pointer assignments in program (after conversion to single dereference form)
- Form set of points-to graph nodes from pointer variables/fields and variables (in the heap or whose address has been taken)
  - Examine each statement, in arbitrary order, and construct points-to edges
    - Merge nodes (and edges) where indicated by unification constraints (only 1 out edge labelled * per pointer variable)
- After (almost) linear pass over these assignments, points-to graph is complete

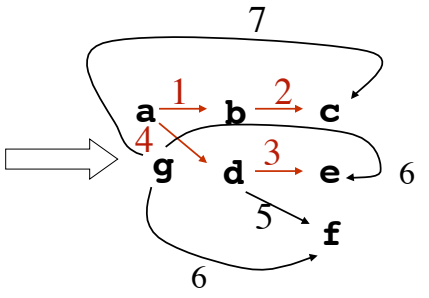
Points-to Analysis for C

- Andersen's analysis (Ph.D.Thesis 1994)
  - Uses inclusion constraints so that for pointer assignments, p = q, algorithm makes

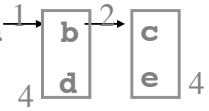
    Pts-to(q) $\subseteq$ Pts-to(p)
  - Points-to graph is larger (i.e., has more nodes) than Steensgaard's and more precise
  - Cubic worst case complexity in program size, $O(n^3)$
  - One points-to set per pointer variable over entire program

Andersen - Example

```
int **a;
int *b,*d,*g;
int c,e,f;
1.a = &b
2.b = &c
3.d = &e
4.a = &d
5.d = &f
6.g = d
7.g = *a
```



Steensgaard solution



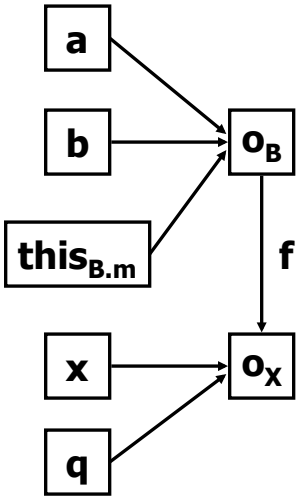
Andersen's Solution Procedure - At a glance

- Find all pointer assignments in program
- Form set of points-to graph nodes from pointer variables/fields and variables on the heap or whose address is taken
  - Examine each statement, in arbitrary order, and construct points-to edges
    - Need to create more edges when see p = q type assignments so that all outgoing points-to edges from q are copied to be outgoing from p (i.e. processing inclusion constraints)
    - If new outgoing edges are added subsequently to q during the algorithm, they must be also copied to p
    - Work results in $O(n^3)$ worst case cost
  - Treat parameter - argument associations like assignment statements

Example of Points-to Analysis

```
class A { void m(X p) {..} }
class B extends A {
    X f;
    void m(X q) { this.f=q; }
}
```

```
B b = new B();
X x = new X();
A a = b;
a.m(x);
```



Note: A.m() not analyzed because it's unreachable.

Constraints Generated

- B b = new B(); $\{o_B\} \subseteq PtsTo(b)$
- X x = new X(); $\{o_X\} \subseteq PtsTo(x)$
- A a = b; $PtsTo(b) \subseteq PtsTo(a)$
- a.m(x);
  - Arg-param relations cause: $this_m$ = a; q = x; which generates: $PtsTo(a) \subseteq PtsTo(this_m)$, $PtsTo(x) \subseteq PtsTo(q)$
- Then we process the code within m()
  - $this_m$ .f = q
- *A satisfying assignment for these constraints is a points-to solution for this code.*

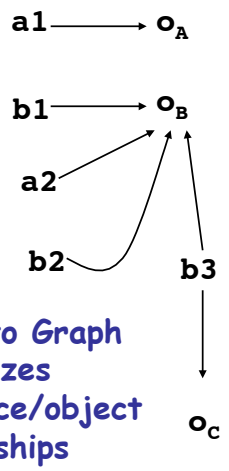
FieldSens Points-to Analysis for Java

- Based on Andersen's points-to analysis but also add object reference fields to points-to graph as suffices for reference variables
  - e.g., class A has fields f, g then p=new A(), means p.f and p.g are in the points-to graph
- Define and solve a system of annotated set-inclusion constraints
  - Handles virtual calls by simulation of run-time method lookup
  - Models the fields of objects
  - Extended BANE (UC Berkeley) constraint solver
- Analyzes only possibly executed code
  - Ignores unreachable code from libraries

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints" OOPSLA'01

FieldSens Example

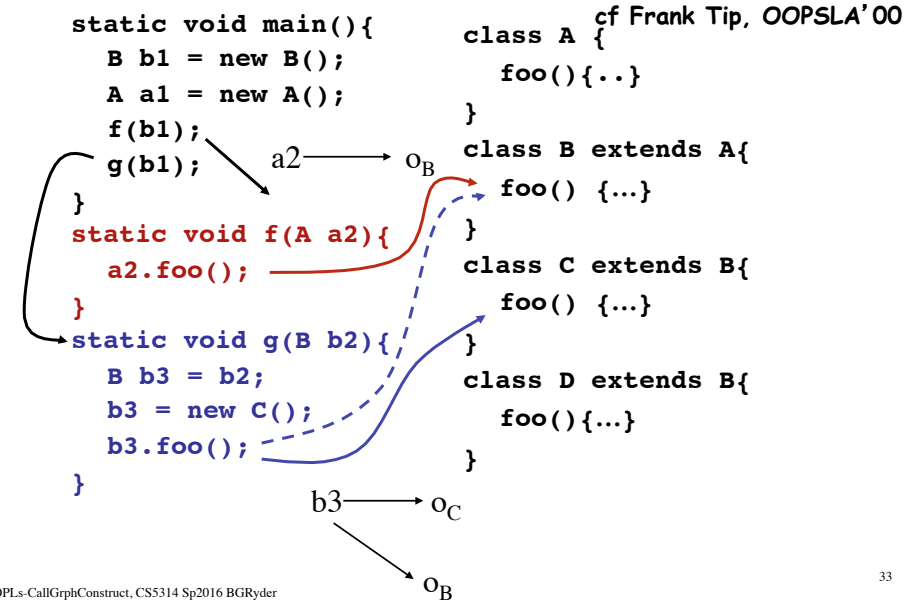
```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```



Points-to Graph summarizes reference/object relationships

FieldSens Example

cf Frank Tip, OOPSLA'00



FieldSens Characteristics

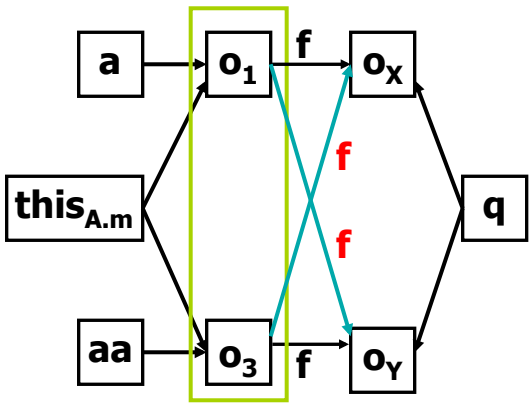
- Only analyzes methods *reachable* from main()
- Keeps track of individual reference variables and fields
- Groups instances of objects by their creation site
- Incorporates reference value flow in assignments and method calls

FieldSens Findings

- Empirical testing found
  - Significant precision gains over RTA at call sites found to be polymorphic by CHA
  - Generated useful points-to info for client analysis
    - Object read-write information
    - Synchronization removal (thread-local)
    - Stack allocation (method-local)

Imprecision of Context Insensitivity

```
class Y extends X { … }
class A {
  X f;
  void m(X q) {
      this.f=q  ;  }
  }
A a = new A()  ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```

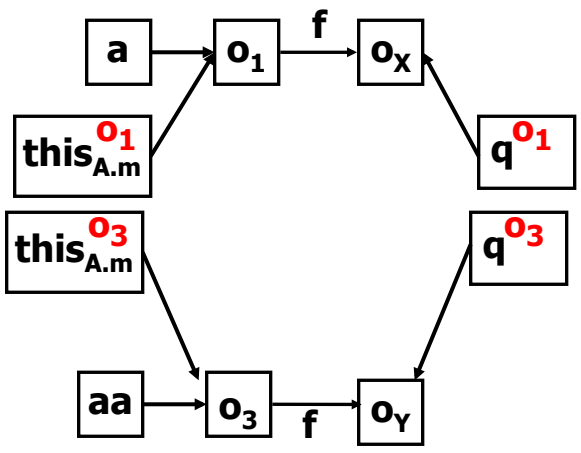


Object-sensitive Analysis

- A kind of functional context sensitivity for flow-insensitive analysis of OO languages
- Formulate an object-sensitive Andersen's (points-to) analysis
  - Analysis of instance methods and constructors distinguished for different contexts
  - Receiver objects used to distinguish calling contexts
  - Empirical evaluation vs. context-insensitive FieldSens analysis
    - `this`, formals and return variables (effectively) replicated

Example: Object-sensitive Analysis

```
class A {
  X f;
  void m(X q) {
this^o3_A.m.f=q^o3; }
  }
A a = new A();
a.m(new X());
A aa = new A();
aa.m(new Y());
```



ObjSens Findings

- Precision gains for problems such as def-uses for object fields and side effect analysis (per statement) for practically no additional cost
- Clients
  - Program test coverage metrics
  - Program slicing
  - Program understanding tools

A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.Followup TOSEM paper 2005.